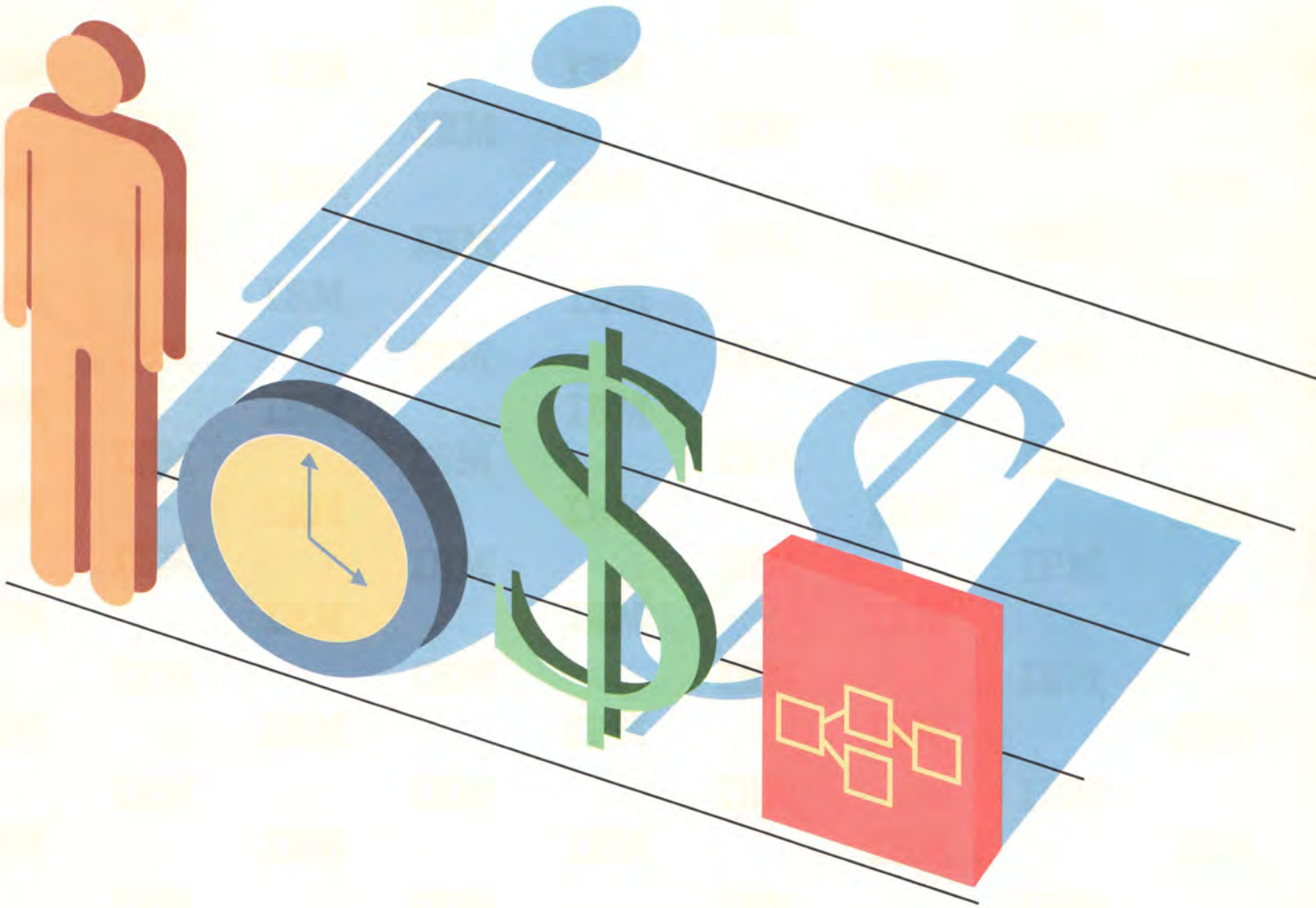
$9.95 USA

IBM Personal Systems Developer

A Publication of the IBM Developer Assistance Program
No. 1, 1992

- OS/2 2.0 Controls
- Printing and Fonts
- Spotlight on Metaphor



IBM Personal Systems Developer

Table of Contents

Winter 1992, No.1

The *IBM Personal Systems Developer* is published quarterly by IBM Software Developer Support, Internal Zip 2230, 1000 NW 51 Street, Boca Raton, Florida 33429. Phone (407) 982-6408, FAX (407) 443-4233. IBM employees and branch office customers can subscribe to the *Personal Systems Developer* through IBM Mechanicsburg's Systems Library Subscription Service (SLSS) using the *Developer's* order number, G362-0001. Others can subscribe by calling the publisher, Graphics Plus Inc., directly at (800) READ-OS2. Subscriptions (U.S. only) are $39.95 yearly. International subscriptions are also available from the Publisher, phone (203) 366-4000, FAX (203) 368-9100. Questions, suggestions and article ideas should be sent to the Editor.

While back issues are not generally available, articles from the first seven issues of the *Personal Systems Developer* have been published in the *OS/2 Notebook: The Best of the IBM Personal Systems Developer*. This book can be bought at a local bookstore ($29.95) or by calling Microsoft Press at (800) MS-Press. Its Mechanicsburg order number is G362-0003-00.

Editor: Dick Conklin, IBM Software Developer Support. Publisher: Graphics Plus, Inc. 640 Knowlton Street, Bridgeport, CT 06608, Project Manager: Jo-Ann Radin-Campbell.

IBM Personal Systems Developer is published by the Entry Systems Division of International Business Machines Corporation, Boca Raton, Florida, U.S.A., Dick Conklin, Editor.

IBM believes the statements contained herein are accurate as of the date of publication of this document. **However, IBM hereby disclaims all warranties either expressed or implied, including without limitation any implied warranty of merchantability or fitness for a particular purpose. In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damage arising out of the use or inability to use any information provided through this publication even if IBM has been advised of the possibility of such damages, or for any claim by any other party.**

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This publication may contain technical inaccuracies or typographical errors. Also, illustrations contained here may show prototype equipment. Your system configuration may differ slightly.

This publication may contain articles by non-IBM authors. These articles represent the views of their authors. IBM does not endorse any non-IBM products that may be mentioned. Questions should be directed to the authors.

This information is not intended to be an assertion of future action. IBM expressly reserves the right to change or withdraw current products that may or may not have the same characteristics or codes listed in this publication. Should IBM modify its products in a way that may affect the information contained in this publication, IBM assumes no obligation whatever to inform any user of the modification.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming or services in your country.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

To correspond with the *IBM Personal Systems Developer*, please write to the Editor at IBM Corporation, Internal Zip 2230, P.O. Box 1328, Boca Raton, FL 33429-1328.

Trademarks

IBM, Operating System/2, OS/2, OS/2 EE, Audio Visual Connection, AIX, AS/400, Application System/400, Operating System/400, OS/400, Personal System/2, PS/2, Writing to Read, SQL/400, Proprinter, Micro Channel, Systems Application Architecture, Presentation Manager, and AT are registered trademarks of IBM Corporation.

DATABASE 2, DB2, CICS/MVS, SAA, Structured Query Language/Data System, SQL/DS, Common User Access, CUA, Dialog Manager, Enterprise System/9370, Common Programming Interface, Distributed Automation Edition, DAE, ES/9000, CICS OS/2, System/370, System/9000, Repository Manager, Enterprise System/9370, VM, RISC System/6000, System Performance Monitor/2, Paperless Manufacturing Workplace (PMW), Print Manager, PlantWorks, Print Manager, IBM4019, and Plant Floor Series are trademarks of IBM Corporation.

Ethernet is a trademark of Xerox Corporation.

Pagemaker is a registered trademark of Aldus Corporation.

Microsoft, CodeView, MS-DOS, Excel, and MS are registered trademarks of Microsoft Corporation.

Windows is a trademark of Microsoft Corporation.

Macintosh and Apple are registered trademarks of Apple Computer, Inc.

COMDEX is a trademark of the Interface Group, Inc.

OpenLook and AT&T are registered trademarks of American Telephone and Telegraph Corporation.

C++ version 2.1 is a trademark of American Telephone and Telegraph Corporation.

Smalltalk is a trademark of Digitalk, Inc.

FingerTips is a trademark of Fortis Development Corporation.

PKZIP2 is a trademark of PKWARE Inc.

PostScript, Adobe Type Manager and Adobe are registered trademarks of Adobe Systems Inc.

UNIX is a registered trademark of Unix Systems Laboratories, Inc.

Lotus and 1-2-3/G are registered trademarks of Lotus Development Corporation.

Turbo Pascal is a registered trademark of Borland International, Inc.

386 and 486 are trademarks of Intel Corporation.

Metaphor and Capsule are registered trademarks of Metaphor Computer Systems, Inc.

TopSpeed is a registered trademark of Jensen & Partners, Inc.

Hewlett-Packard and Laserjet are registered trademarks of Hewlett-Packard Company.

Hewlett-Packard 7550A is a trademark of Hewlett-Packard Company.

CompuServe is a registered trademark of CompuServe Inc.

Editor's Comments

In this issue we have several articles about new features in OS/2® version 2.0. Many of these articles were written by people in IBM's PM Extensions department in Cary, NC. For those of you who thought that all of OS/2's development was done in Boca Raton, and never heard of Cary, we've provided a map.

CARY WHO?

Back in May 1989, Cary's Common User Access™ (CUA™) organization became part of a multi-group project charged with creating new CUA graphic components for OS/2's Presentation Manager.® The Cary team advocated the idea of embedding code for new CUA controls and protocols directly in the operating system. They presented their idea to the OS/2 development organization which led to the creation of the PM Extensions department. The department was initially staffed with nine programmers. The first CUA controls, Spinbutton and Direct Manipulation Protocol were released in December 1990 in OS/2 SE and EE 1.3. The group has since developed six additional controls which will be part of OS/2 2.0: the Container, Notebook, Slider, Value Set, Font and File Dialogs. These controls, described in this issue, allow application developers to implement CUA '91 using OS/2 2.0.

SPOTLIGHT ON METAPHOR

This issue's Spotlight feature is on Metaphor,® a company very much in the news lately. The people at Metaphor were pioneers in graphical user interfaces, client-server architecture, and 32-bit operating environments. So, it's no surprise that they were one of the first companies to migrate to OS/2 version 2.0. Metaphor has also played a major role in the recent Patriot Partners and IBM-Apple® ventures. We'll look at this innovative company and its popular Data Interpretation System product.

WHAT'S YOUR STORY?

We have been writing about OS/2 version 2.0 since our Winter 1990 issue. We monitored the evolving product and brought you early previews of the Software Developer's Kit, the new 32-bit API, software tools, DOS support, math co-processor support, named pipes, thunking, performance tuning, semaphore APIs, and more. Our intention is to continue bringing you the latest news and practical advice on getting the most out of OS/2 2.0. This means more technical articles from IBM's OS/2 developers. It also means more case studies, tips and techniques from pioneering OS/2 application developers.

Dick Conklin

Do you have a story to tell or a good idea to share? We would like to hear from you. The process is simple:

1. Fax a short outline of your proposed article to me at (407) 443-4233.
2. I'll fax back a copy of our article submission guidelines.
3. If we like your idea, we'll work with you to get it published in the *Developer*.

Dick Conklin

Our intention is to continue bringing you the latest news and practical advice on getting the most out of OS/2 2.0

Developer Assistance Program

Developer Assistance Program Update

Joe Carusillo

OS/2 2.0 EARLY CODE PROGRAMS

The Developer Assistance Program has three separate early code programs for OS/2® 2.0.

The first program is the OS/2 2.0 Compatibility Test Program. It is for developers who want to test their existing DOS, Microsoft® Windows,™ and 16 Bit OS/2 products with OS/2 2.0. It includes a beta version of OS/2 2.0 and the option to receive the IBM OS/2 LAN requester and/or the Novell® Netware Requester for OS/2. To date, we have delivered more than 2,000 copies of OS/2 2.0 early code under this program. As a result thousands of existing software products will have been compatibility tested with OS/2 2.0 by the time it becomes generally available. If you need more information on this program contact us through the DAP hotline at (407) 982-6408.

The second is the OS/2 2.0 32-Bit Expedite Program. This program provides a complete beta code developers kit for producing new 32-bit specific OS/2 applications. It also provides a priority technical support channel via MCI Mail to assist you with any questions you have in using it. The developers kit includes beta versions of: OS/2 2.0 Operating System; OS/2 2.0 Toolkit including online documentation of API's, for the base operating system and the Presentation Manager; IBM C Set/2 (IBM's "C" Compiler and PM Debugger); WorkFrame/2 (an OS/2 application development environment); a Windows to OS/2 migration kit; Workplace Shell API documentation; and the System Object Model (SOM) Guide and Reference.

To enroll in the 32-Bit Expedite Program or to receive more information on it, contact the IBM 32-Bit Expedite Distribution Center at 1-800-IBM-3040.

The third program is the IBM Early Application Development Program. For a one time program fee of $750.00, members will receive the developer kit from the expedite program, plus copies of the IBM and Novell Prerelease OS/2 requesters, plus a final shrink-wrap copy of OS/2 2.0, the OS/2 2.0 Toolkit, IBM C Set/2, and IBM Workframe. Enrollment in the IBM Early Application Development Program is open to all OS/2 developers who join before March 1, 1992. Members will be required to fill out and return the feedback form supplied in the offering. To enroll in the program or to receive more information on it, contact us at 1-800-IBM-3040.

INTERNATIONAL TOOLS CONFERENCE

The first OS/2 2.0 International Tools Conference was held in Ft. Lauderdale in November. It was a tremendous success. Over 70 software development tools for OS/2 were displayed. The 600+ attendees had 40 different breakout sessions to choose from. Lee Reiswig gave his OS/2 Live presentation with a large contingent from the Boca Raton Programming Center on hand to give some of the presentations and answer questions. As part of the registration fee, all attendees received copies of the latest OS/2 2.0 beta code. In the next issue, look for a schedule of the conferences planned for 1992.

SERVICES UPDATE

The DAP hardware rebate program has been enhanced recently. The rebate percentage has been increased from 10 to 15% on most PS/2 386/486 class systems and selected PS/2 options. Current members will receive a new list of eligible hardware indicating which items qualify for the new 15% rebate.

The new Common User Access™ (CUA™) design books are now available through PC Books and Special Promotions. Available books include: *Common User Access: Guide to User Interface Design, Common User Access: Basic Interface Design Guide, and Common User Access: Advanced Interface Design Reference.* For more information please contact us at the DAP.

DAP SUPPORT IN CANADA

Since January 1991, a Developer Assistance Program has been available in Canada. Like its U.S. cousin, its mission is to provide assistance to Canadian software developers working on producing new products for OS/2. The eligibility requirements for membership are the same as in the U.S. The services offered include: technical support, IBMLink, OS/2 2.0 Early Code, discounts on IBM hardware and software, and complimentary copies of the *PS Technical Solutions Journal* and the *IBM Personal Systems Developer.*

In addition, ESDTOOLS is available to Canadian DAP members. Like the U.S. version, there is a separate license agreement that needs to be completed prior to being granted access to it. For more information on ESDTOOLS or about the DAP in Canada call (416) 946-3776.

OS/2 APPLICATION SOLUTIONS DIRECTORY

In October, the first issue of the *OS/2 Application Solutions* directory was published. This directory lists more than 1400 applications that are available for OS/2. It contains complete program descriptions including order information and company contact. The OS/2 Application Solution directory retails for $19.95 and is available direct from the publisher, Graphics Plus, by calling 1-800-READ-OS2.

ELIGIBILITY AND ENROLLMENT

The IBM Developer Assistance Program is for software developers working on products for commercial release. There is no charge for membership. Participation is open to developers who develop products that support IBM OS/2 and are currently marketing a PC software product.

If you are not currently marketing a product, you can submit a non-confidential, detailed business plan showing marketing and development activities and schedules. IBM reserves the right to accept or reject an application based on this business plan. To request an application form or to ask questions about any of our programs contact:

IBM Corporation — DAP
Internal Zip 2230,
P.O. Box 1328
Boca Raton, FL 33429-1328
(407) 982-6408

Tricia Kelley, IBM Austin demonstrating Database Manager, a component of Extended Services at the Tools Conference held in November.

Tony Fazio, president Fortis Development, discussing FingerTips™ a real-time OS/2 application development environment at the recent Tools Conference held in November, *see page 58 for more details.*

CORRECTION

In the last issue the new price of a seat in one of the Migration Workshops was reported to be $2,000.00. In actuality it is $2,300.00. I regret any confusion and inconvenience this has caused anyone.

Joe Carusillo, *IBM Personal Systems, 1000 NW 51st Street, Boca Raton, FL 33431. Mr. Carusillo is the Manager of Software Developer Programs which includes the Developer Assistance Program. He started with IBM Entry Systems in 1982 and has worked on PC software since that time. He has a BS in Computer Systems Engineering from the Columbia University, School of Engineering and Applied Science, in the City of New York.*

Spotlight on Metaphor

Metaphor and the IBM Data Interpretation System

Gary Muller

by Gary Muller

It has been 10 years since the IBM® Personal Computer was introduced. Yet, repeated surveys of Fortune 500 PC users confirm that, typically, the only programs being used extensively within the business community remain spreadsheet and word processing programs. These applications were the catalyst for the early growth of the PC industry, but it was expected that they would dominate the market only through the early 1980s. However, these programs still dominate the market despite the fact that new applications, such as desktop publishing, expanded the potential computer user base in the late 1980s. Many software developers continue to add features to these basic applications, trying to capture a larger share of the same users instead of developing new applications that would expand their markets. Growth and innovation seem to have been stalled by the illusion that other products were just around the corner, from "somewhere." (See Figure 1.)

This month's Spotlight article is about a software application that is focusing on this need for other applications from "somewhere." Metaphor® Computer Systems, Inc., a wholly-owned subsidiary of the IBM Corporation, is a unique company whose products and services provide advantages to Fortune 500 professionals who have the need to access diverse sources of information quickly and use that information to make efficient and timely decisions. Metaphor's products are installed in approximately 150 Fortune 500 companies in more than 17 industries. Their keystone product is the IBM Data Interpretation System (DIS) illustrated in Figure 2.

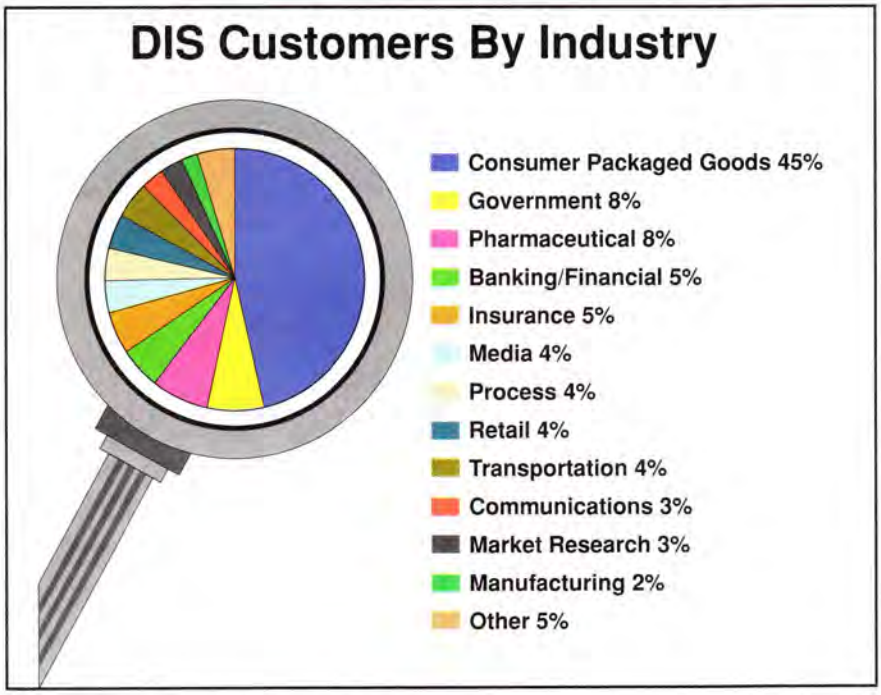


Figure 1. Chart of Current DIS Usage by Industry

COMPANY BACKGROUND

Metaphor was founded in 1982 by David Liddle and Don Massaro, who had worked together previously at Xerox Corporation. During his 10 years at Xerox, Liddle was at the corporation's Palo Alto Research Center, where he did early research on graphical user interface design. He later served as vice president and general manager of the Xerox Office Systems Division. Before co-founding Metaphor, Massaro was president of the Xerox Office Products Division. Today, Metaphor employs over 400 people and is headquartered in Mountain View, California. A privately-held company until October, 1991, it has 13 sales and support offices in major metropolitan areas throughout the United States and in London.

According to Charles Irby, Chief Technical Officer, the company has issued 15 major product releases in the last seven years, with the more recent ones being specifically IBM-based. He adds, "We have a more substantial development organization than you might expect because of the complexity of our product (DIS). It touches almost every aspect of an enterprise's computing environment with a lot of LAN, host, and PC expertise blended together into one complete system."

Charles Irby

Many applications claim to have easy-to-use data access, but Metaphor's DIS lives up to its claim. One reason is a consistent user interface. Metaphor places considerable emphasis on having a consistent interface — one that has a "minimum number of ways to accomplish any given task." Irby also states that the company focuses on performance because the user interface should respond to interactions very quickly. According to Irby, Metaphor has developed a total of 45 tools so far, all of which work very well together. For example, it is easy to transfer data from one tool to another because of the architecture of the system, an architecture that was conceived early in the product development cycle.

Metaphor has been a leader in applying object-oriented technology to real business issues. In DIS, a single icon represents both a procedure (or tool) and the associated data. Complex processes can be performed with a single action that seems extremely simple from the user's perspective. For example, the Query icon transparently invokes procedures for connecting to a database independent of its type or physical location. When the database is opened, the Query tool knows exactly how to make and manage the database connection and can do so in a way that is transparent to the user.

When introduced in 1984, Metaphor's DIS ran on a hardware platform of the company's own design. At that time, existing desktop platforms lacked the processing power to support software as robust as DIS. Therefore, from 1984 to 1988, DIS was sold exclusively on Metaphor's own hardware.

Metaphor's original products were based on Motorola's 32-bit MC68000 microprocessor technology. When PC 32-bit technology was introduced, the company developed software based on Intel's 32-bit 80386 chip.

IBM, aware of Metaphor's success with Fortune 500 customers, recognized DIS as a software product that could help propel both its PS/2® products and the OS/2® operating environment throughout its own customer base. In 1988, IBM licensed the PS/2-based product and technology from Metaphor to provide a Data Interpretation System based on PS/2, OS/2, and Micro Channel® technology. The resulting product was called the IBM Data Interpretation System (DIS).

By 1990, IBM and Metaphor expanded their agreement to include the joint marketing of a single DIS product, thereby migrating all functions of the Metaphor product line to the IBM PS/2. With this agreement in place, and with industry-standard hardware already running DIS, Metaphor announced that it would discontinue the development and manufacturing of hardware and would concentrate on providing DIS and other innovative software products for industry-standard platforms.

Later in 1990, IBM and Metaphor intensified their link by forming a joint venture, Patriot Partners, to develop advanced object-oriented system software. In July of 1991, IBM agreed to buy Metaphor, which would operate as a wholly-owned subsidiary.

WHAT IS THE IBM DATA INTERPRETATION SYSTEM?

Choosing the right decision support system is a challenge. Decision makers want to ensure that they are able to access data and turn it into information that supports better and more timely decisions. To be more effective, they need a solution that they can use to develop their own unique applications. However, they still need to manage their access to data through a system that can become an integral

It (DIS) touches almost every aspect of an enterprise's computing environment...
— Charles Irby

part of their overall information system architecture. Metaphor and the IBM Data Interpretation System (DIS) can help them meet these needs.

What sets DIS apart from other products? Much of what is different about DIS is technical in nature. Another difference is this: when customers choose DIS, they not only select a product, they also select Metaphor's expertise and desire to be a partner with them to help solve their business problems. DIS also shifts the balance in terms of both how much time is spent collecting data and how much time is spent analyzing the data that has been collected. Put simply, productivity increases.

DIS is made up of application "tools" that are highly integrated and allow data to flow seamlessly through access, analysis, presentation, and sharing functions. Data can be accessed from relational databases over local or wide area networks, but users are shielded from the underlying Structured Query Language (SQL) and intricacies of network access. Traditional computing environments require users to memorize command languages, but DIS users need only recognize the appropriate activity for the intended analysis in order to access and organize data quickly and easily.

DIS gives business professionals a complete, integrated set of software tools to help them make strategic use of data: query tools and report writers for direct and easy data access; spreadsheet, plot, and statistical tools for analysis; as well as capabilities for sharing analyses and applications with business team members across the hall or around the world. And with the unique Capsule® facility, users can link individual tools together in seconds to create their own push-button applications (see Figure 2).

Accessed data can be structured and analyzed quickly in a spreadsheet tool and then displayed in a plot tool that includes a wide choice of graphics such as bar, pie, and scatter charts. The process is so smooth and fast that users can readily take several different "cuts" at the data to find interdependencies. To help make a repeated analysis even more efficient, any DIS process can be captured as a Capsule application and subsequently invoked again with just one or two clicks of a mouse. In essence, this allows users to create their own tailored applications without programming, simply by encapsulating a sequence of tools. Capsule applications also can be modified easily and can be shared with other users.

You don't program in the strictest sense; you take objects and visually connect them to show the way that data flows through an analysis process

—Karen Pasternack Straus

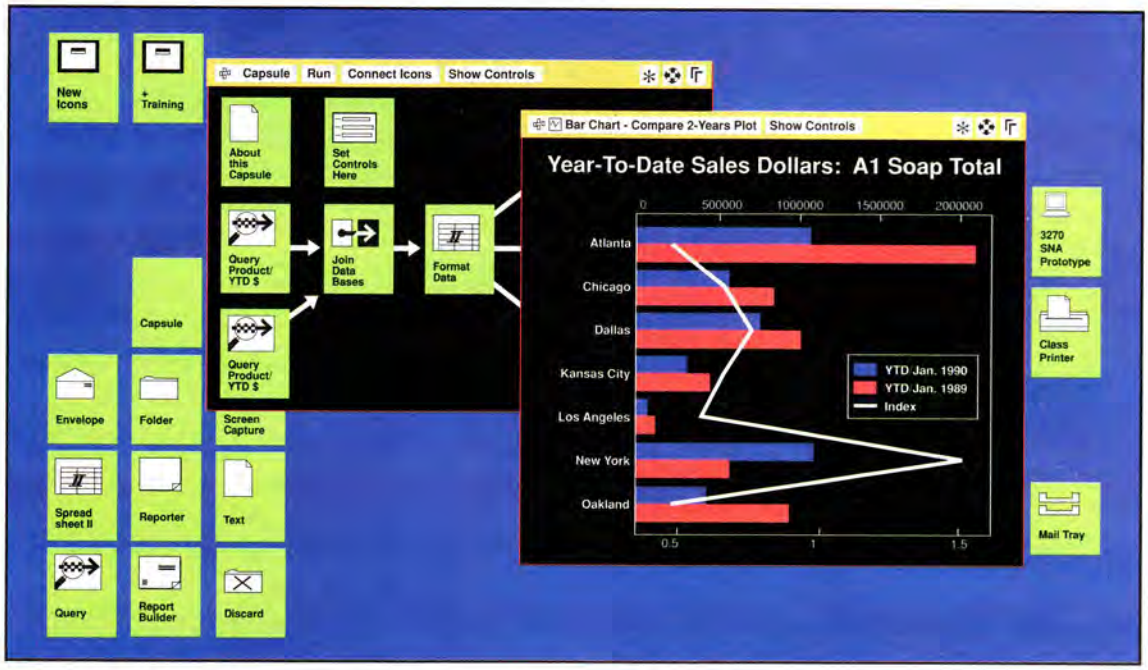


Figure 2. DIS Desktop With Open Capsule Application and Graph Results Windows

Karen Pasternack Straus

Karen Pasternack Straus, Manager of DIS Product Management, notes that the use of the Capsule tool also assists the programming effort. She explains, "You don't program in the strictest sense; you take objects and visually connect them to show the way that data flows through the analytic process."

As an example, a simple Capsule application might include a Query tool, a Spreadsheet tool, and a Plot tool. You would use the Query tool, with its visual interface, to build an "ad hoc" query to compare one thing with another. The Query results could be output to a spreadsheet. You then would set up the regions and formulas in the spreadsheet to do some additional analysis. Finally, just by using the point-and-click interface, you could drop the results into the Plot tool. With the Plot tool, the analyst has a very visual picture of the data analysis upon which to make a decision. "You can save this process and rerun it with new data simply by dropping the tools into the Capsule and then drawing an arrow from the Query tool to the Spreadsheet tool and from the Spreadsheet tool to the Plot tool. You have now programmed an application," says Pasternack Straus.

DIS was designed so that non-technical people could create and share their analyses within an organization. And it was designed to work with many of today's strategic data processing environments. It provides access to data through its use of both relational database technology and high-level SQL which are becoming new standards for information management. An application created by the IBM Data Interpretation System accesses data stored in Systems Application Architecture® (SAA™) relational databases, uses distributed local area network (LAN) services, and is written in SAA-compliant languages.

According to Irby, Metaphor feels that both CUA™ '91 and the DIS user interface offer important advantages; therefore, they have been working closely with IBM to modify both so that they eventually will merge into one. Irby says that "it was clear that CUA needed to move in the direction of the DIS user interface as well as the DIS user interface moving in the direction of CUA." He feels, further, that this will facilitate user adoption of object-oriented interfaces.

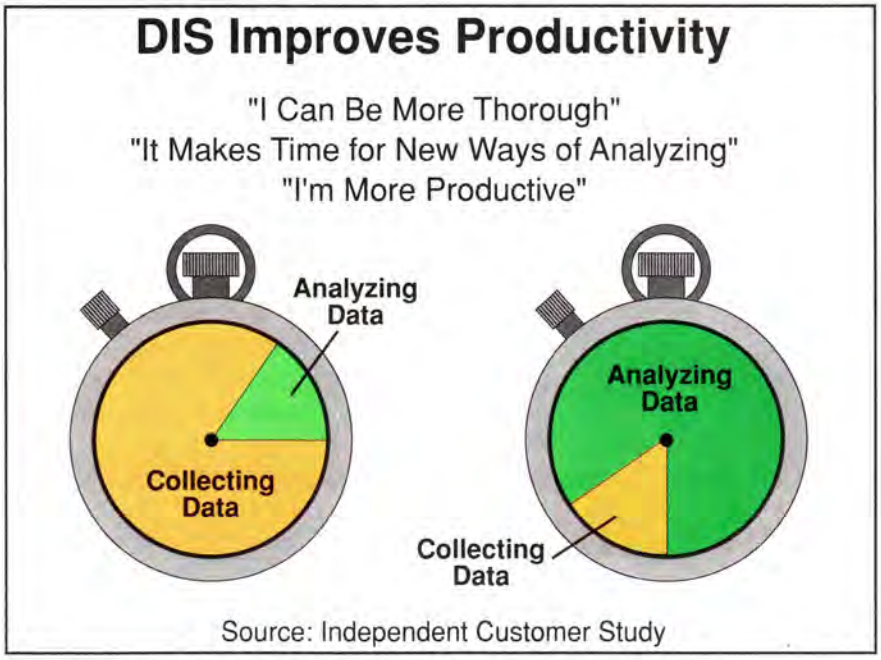


Figure 3. Before and After DIS Usage Productivity Comparison

MARKETING THE PRODUCT

Metaphor is proud of the fact that they are a solutions-oriented company. According to Kathy Mitchell, Vice President of Marketing, "The primary focus is the value that our product brings to the customer." Metaphor learns all about their customers' businesses first, so they can help determine what the customers need in order to be more productive. When the sales force demonstrates the product, they show how it will meet the needs of the customer (see Figure 3).

Kathy Mitchell

According to Pasternack Straus, "We go in and show the customer how DIS provides value. We work with them to produce specific applications and

Unicode is going to become the ASCII of the 1990's
—Charles Irby

user training that will help increase their productivity and, quite possibly, their profitability. We also go back and audit their progress. And we use an 'implementation team' approach."

Metaphor's International Effort

One of Metaphor's directions is to establish DIS as an international product within the next few years. To aid in creating the international version, Metaphor helped define Unicode and was instrumental in forming the Unicode consortium. This consortium was started by a small group of companies that recognized that it was in their best interest to have one, standard "flat" way of representing all of the world's characters. The number of participating companies is growing and, just recently, the ISO 10.646 subcommittee decided to adopt Unicode as a standard. According to Irby, also a Unicode consortium board member, this means that "Unicode is going to become the ASCII of the 1990's." Thus, the foundation has been established whereby different countries could be running different user interfaces in different languages, all using the same underlying character set encoding.

DESIGNING A NEW PRODUCT

When Metaphor begins to work on a new release, it looks at many different areas (customers, competition, technology, and programming team skills, among others) in order to design the most effective product for the largest number of users. Metaphor relies on both market research and specific customer feedback when prioritizing enhancements to the product. They examine trends in technology to identify those that would add the most value for their DIS users — often pushing the limits of a technology in order to achieve the desired result. They organize into project teams and combine the skills of programming, marketing, documentation, and testing so each discipline can contribute to an innovative development approach.

At its lowest level, the DIS development group is organized into functionally oriented teams that work either on individual tools or on specific system service areas. These teams have responsibility from the beginning of a project. They are in place from initial conception to delivery in the field, and then they stay on to support the product.

Metaphor is developing the capability for interoperation between their environment and the OS/2 Workplace Shell environment on OS/2 2.0. This environment may be running other applications on the same machine for the same user. It will be possible to exchange data interactively among DIS, OS/2 and Windows™ applications through the Clipboard. It also will be possible to use these applications plus standard PC applications in a Capsule tool, thus extending the value of this unique product by supporting a wide range of applications. "We see this as a cornerstone in being able to utilize non-DIS applications in the midst of a DIS Capsule," says Irby.

Working with OS/2 2.0

Metaphor is one of the first companies to use the 32-bit capabilities of OS/2. The company has been doing development on 32-bit OS/2 for two years now and is quite pleased with the results. Metaphor plans to exploit the 32-bit interface fully and is confident that OS/2 will become a standard for its customers.

We discussed OS/2 with two DIS OS/2 developers, Steve Lienhard and Bill Malloy.

Steve Lienhard

They indicated that around 20 programmers are working on DIS for OS/2 2.0. Steve Gold and Karen Pasternack Straus of DIS Marketing shared the feeling that the move to OS/2 2.0 is both evolutionary and revolutionary for them. Because there was no appropriate technology at the time, DIS started with its own operating system. As OS/2 2.0 is becoming a richer product, they are now able to begin to phase out their own proprietary operating system and replace it with OS/2 2.0.

The biggest challenge during the development of the OS/2 2.0 version of DIS has been working with early releases of beta code. DIS

is so rich in functionality that seeing all of those functions work on the new platform required waiting until OS/2 2.0 was fully developed, debugged, and stabilized. Gold described the workstation evolution as shown in Figure 4.

Bill Malloy

The port of the DIS servers to OS/2 has also been an evolutionary one. Metaphor began its OS/2 support by absorbing the communications server into OS/2 components and then porting the database server from the 16-bit kernel to the 32-bit kernel. The file server is the central component in the Metaphor design, and Metaphor workstations were built without disk storage. The design perspective at the time was that if you bought good central disk drives and had a fast network such as Ethernet,™ you didn't need local disks and, in fact, were better off without them. Everything was stored on the file server. Whenever you loaded a program, accessed data, etc., you always went back to the file server. But with today's powerful PS/2 workstation, the DIS design has changed as shown in Figure 5.

From the beginning, the product was designed with networks in mind. A key element in the DIS design is database service. For example, it does not matter whether a database is local on a host or on a remote network. "DIS," says Gold, "is a very early example of a client-server network application. When DIS was first released, there was really nothing available like it and, even today, there are few applications that utilize the network to the extent that we do."

The LANs in the Metaphor architecture use the Xerox Network Service (XNS) which includes internetwork routing. Two or more LANs can talk to each other, and most of the operation is transparent to the user. Now, token rings can be hooked to each other, as well as token rings to Ethernet and Ethernet to Ethernet.

The DIS product is still designed entirely as a client-server application that heavily utilizes the LAN. For workstations running OS/2 2.0, Metaphor is creating a local disk cache that helps LAN performance a great deal. (In the prior system, they loaded any executable software from the file server, and they didn't access the local disk at all.)

Steve Gold

According to Steve Gold, "Now, we are taking advantage of the fact that most PCs have hard drives to reduce the number of times we have to return to the file server. We are now using a local disk as a cache. Our research indicates that this should reduce traffic on the LAN dramatically."

Another OS/2 feature that Metaphor took advantage of in order to implement the disk cache is the dynamic link library (DLL). As part of their local disk cache strategy, DIS developers chose to implement the various

DIS is a very early example of a client-server network application
—Steve Gold

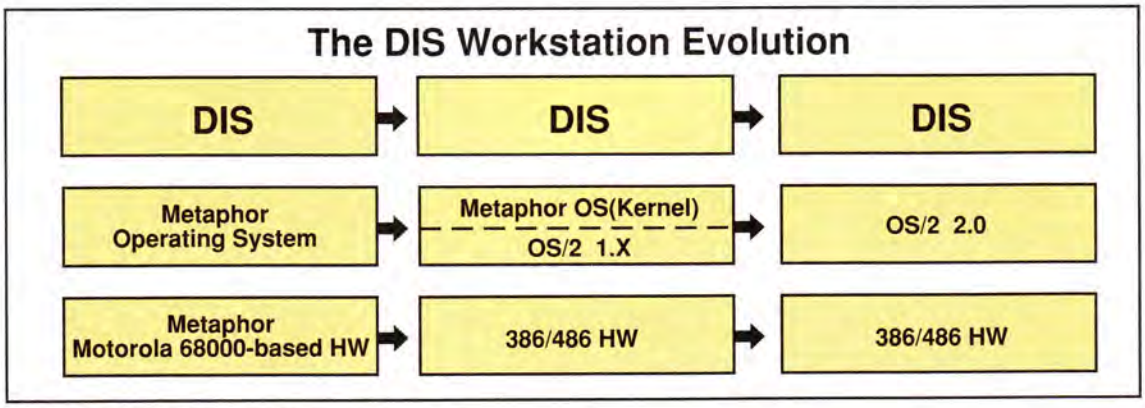


Figure 4. DIS Workstation Evolution

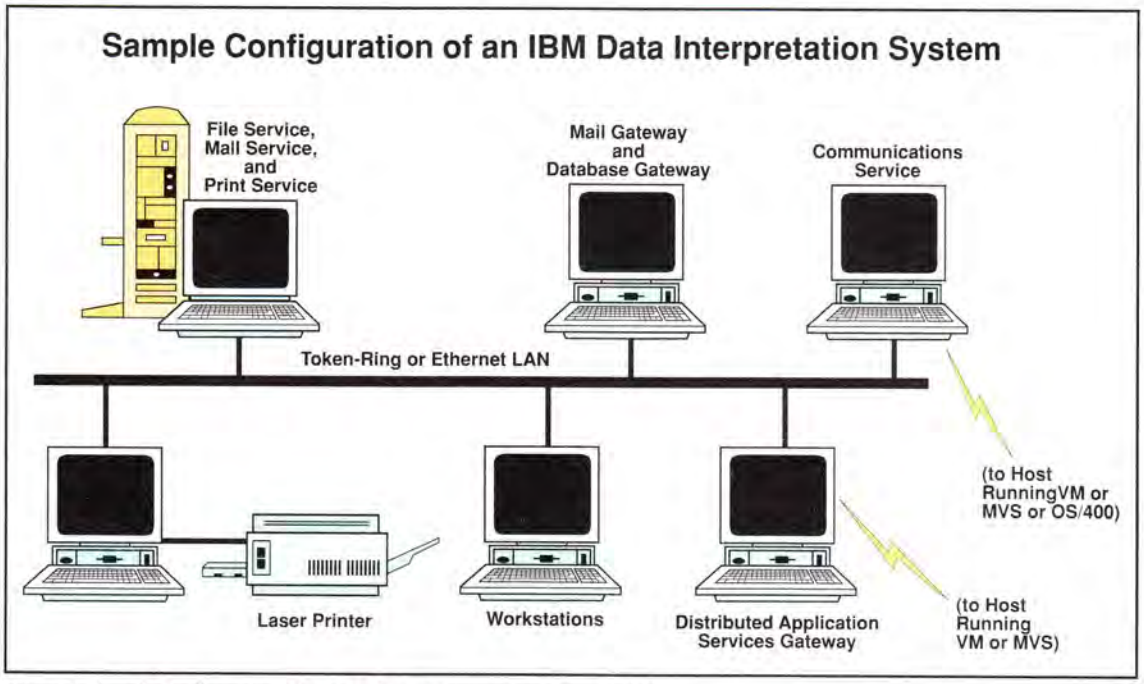


Figure 5. Sample Configuration of an IBM Data Interpretation System LAN

The primary focus is on the value our product brings to the customer —Kathy Mitchell

tools and services that are part of a DIS workstation as DLLs. This enabled them to transfer the necessary DLLs over the network as needed and link to them in real time.

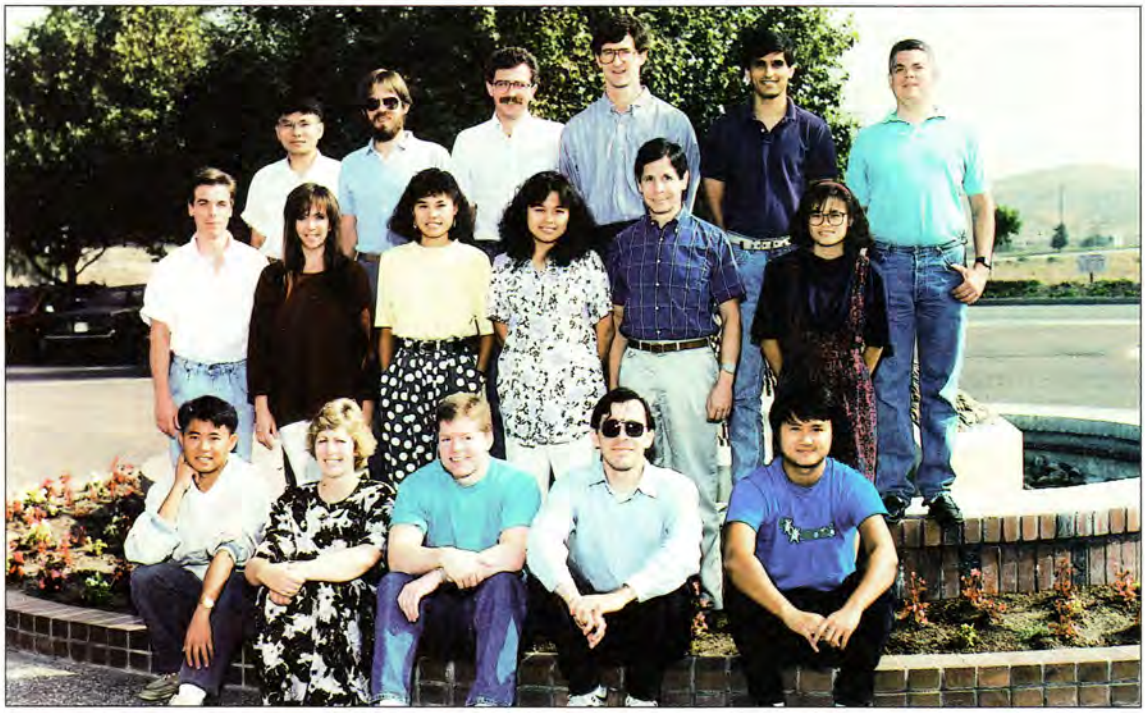
WHAT ABOUT THE FUTURE?

It is obvious that Metaphor is concerned about those people in large corporations who do not use their PCs effectively, either because they are simply too hard to use or because the computer applications on the market do not handle their problems adequately. Charles Irby is concerned that the dynamics of the software industry "are forcing vendors to produce generic software that has wide appeal but cannot be tailored to individual needs." Irby would like to see the industry move toward a more component-like approach where you could combine applications more easily. They could be used standalone or with many different applications. When combined with other applications, these modular products would produce business solutions that would be tailored for specific users. Irby would like to see more attention given to enabling users to adapt applications to their needs without the necessity of getting into programming or even programming concepts.

Metaphor especially wants to reach the 80% of the Fortune 500 professionals who are not using computers. Irby feels this can be done if computer companies like Metaphor are willing to have field organizations made up of sales representatives and consultants who go in and "embrace the problems of the end user and apply technology to solve those problems."

Metaphor also considers the ability of OS/2 2.0 to support 16-bit Windows 3.0 to be extremely important. They expect to see many major corporations employ OS/2 as a common platform for their OS/2 and Windows applications. Irby says that "because of its ability to support Windows, OS/2 will become a much more important product within the next year and a half."

Looking into the crystal ball, Irby also sees the graphical user interface evolving, over the next 10 years, toward a much more object-based interface with heavier use of direct manipulation. "And," says Irby, "I would hope to be able to see a lot more use of animation and three-dimensional renderings that make all of this much more meaningful to the user."



Back row: (standing l-r) Jerry Soung, Paul Stevens, Eli Lauris, Brian Jackson, Sundar Krishnamurthy, and David Morandi. Middle row (standing l-r) Steve Lienhard, Kathy Granlund, Ju-Ju I, Chih-Li I, Peter Herrera, and Donna Lowe. Front row (seated l-r) Winston Tsai, Jacqueline Humfield, Tad Worley, Ken Whistler, and Marc Chiu.

Gary Muller, *IBM Corporation, 1000 NW 51st Street, Boca Raton, FL 33432. Mr. Muller is an advisory information developer in Boca Raton. He joined IBM in Kingston, NY in 1967, where he designed and wrote software manuals for the System/360. Since moving to Boca Raton in 1971, Mr. Muller has written both hardware and software manuals, edited, coordinated projects on information quality, set up customer focus groups and surveys, and managed an information testing department. He is now education coordinator for Information Development. Mr. Muller received an IBM Excellence Award in 1986 for his work on the Personal Computer page design and an Award of Achievement at the 1986 STC Conference for his work on a local style guide. An author and presenter of many courses in technical communication, he also taught technical writing at the College of Boca Raton and is a frequent guest lecturer at the University of Central Florida in Orlando.*

Metaphor Computer Systems, Inc.

Address:	1965 Charleston Road Mountain View, CA 94043 (415) 961-3600
Founded:	1982 (acquired by IBM as a wholly-owned subsidiary October, 1991)
Employees:	457
Sales Channels:	An IBM Program Product, marketed by IBM and Metaphor Direct Sales Forces
Information Contact:	(800) 346-3824

metaphor

32-Bit OS/2

New Controls in OS/2 2.0: An Overview

Peter Haggar

by Peter Haggar

Several articles in this issue discuss function in OS/2 release 2.0. Since OS/2 2.0 was not generally available at the time we went to press, it is possible that some of these features (and others found in beta releases) may not appear in the final product. It is also possible that some additional features may appear in the final product that were not known at press time.

The IBM® Cary PM™ Extensions department is shipping six new application development tools with OS/2® 2.0. These tools, also called PM controls, consist of the following:

- *File Dialog*
- *Font Dialog*
- *Value Set Control*
- *Slider Control*
- *Notebook Control*
- *Container Control*

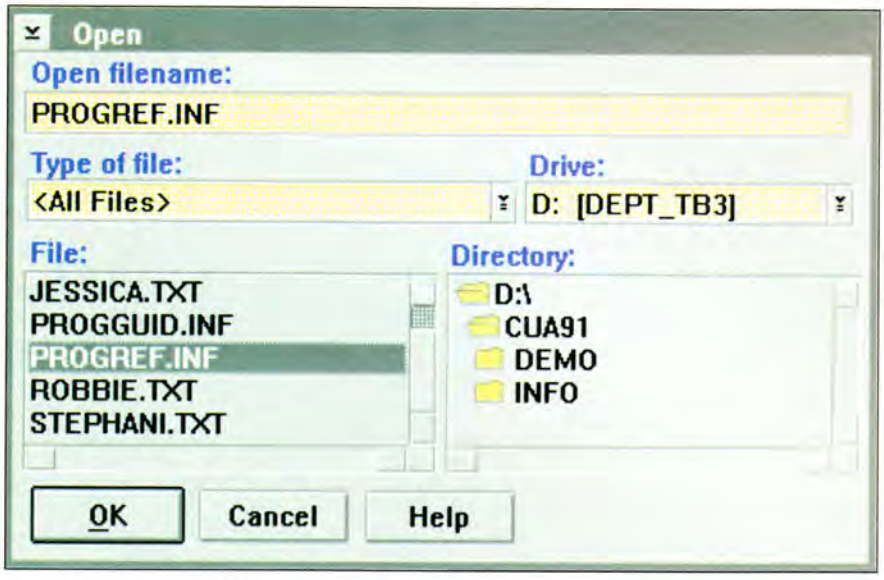


Figure 1. A File Open Dialog

These controls are similar to existing PM controls, (i.e., scrollbar, listbox, spinbutton, etc.) in that they will be part of the operating system with public Application Program Interfaces (API's) and can be used by any application wishing to exploit them. Although each is different, these new controls provide application developers additional power to build CUA-conforming applications.

OVERVIEW

The following is a brief overview of the six new controls. Information is provided about what each control does and how each could be used in your applications. (For more information, please refer to the five articles which follow this one.)

File Dialog

Using the File Dialog, application developers can utilize a simple, consistent user interface for end users to open and save files.

The File Dialog can be implemented as either a file Open or SaveAs. Both dialogs are very similar in their appearance and user interface. They provide an entryfield to enter a specific file name to be opened or saved with wild cards (*) supported. Users can navigate through different drives, directories, and files to locate the file they wish to open or save. Both dialogs also support Extended Attributes (EAs). Once the user has chosen a file to open or save, its file name is returned to the application for processing. The File Dialog can also be customized by an application for its own specific use. Standard controls such as buttons and entryfields may be added.

An example of the File Dialog control can be seen in Figure 1.

Font Dialog

The Font Dialog enables applications to allow users to view and select fonts available for use. The basic functions of the Font Dialog provide the ability for the user to select from a list of font family names installed on the system, styles for each font, the available sizes for each font, and the emphasis styles available for each font.

The Font Dialog also has a viewing area that is updated dynamically when the user is making selections of fonts, styles, sizes, and emphasis. This viewing area is useful to the user in order to preview the font prior to accepting and applying it.

The Font Dialog can also be customized by an application for its own specific use. Standard controls such as buttons and entryfields may be added.

An example of the Font Dialog control can be seen in Figure 2.

Value Set

The Value Set control is similar to the existing Radio Button control since its purpose is to allow a user to select an item or items from an existing set. However, unlike radio buttons, the Value Set provides a graphical set of selectable items. Suppose your application is providing the ability to select 1 of 50 shades of the color blue. Using the Value Set, you have the ability to provide a graphical image of the 50 selectable shades. This would enable the user to see the shade of blue and make a choice based on this. Using radio buttons with textual descriptions of the 50 shades of blue would clearly not be in one's best interest. The Value Set control supports text like a radio button but offers greater flexibility as an application's needs grow and change.

The system drag protocol (provided in OS/2 1.3) is supported by the Value Set Control making it all the more useful. A possible use of the Value Set would allow users to drag the item that represents the shade of blue they want. They could then drop it on the client area of another window to change that window's background color.

An example of the Value Set control can be seen in Figure 3.

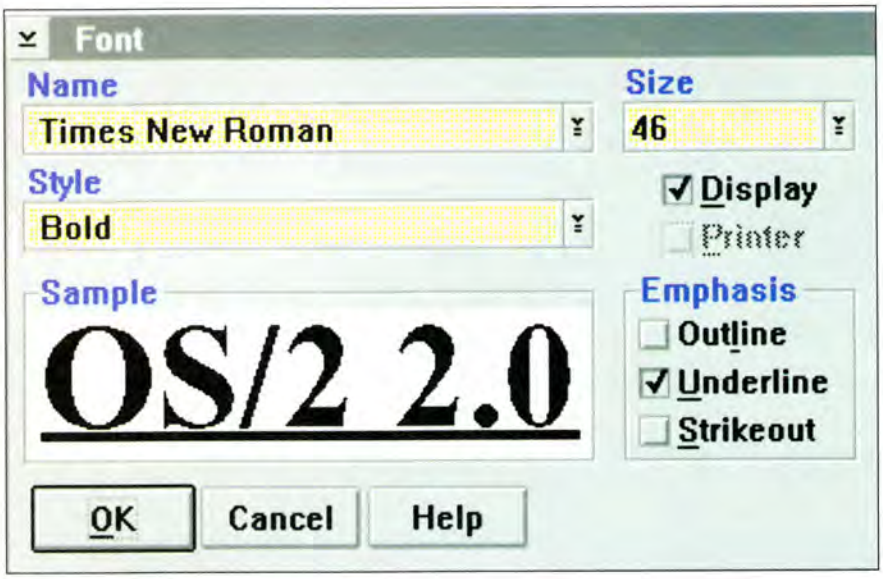


Figure 2. A Font Dialog Control

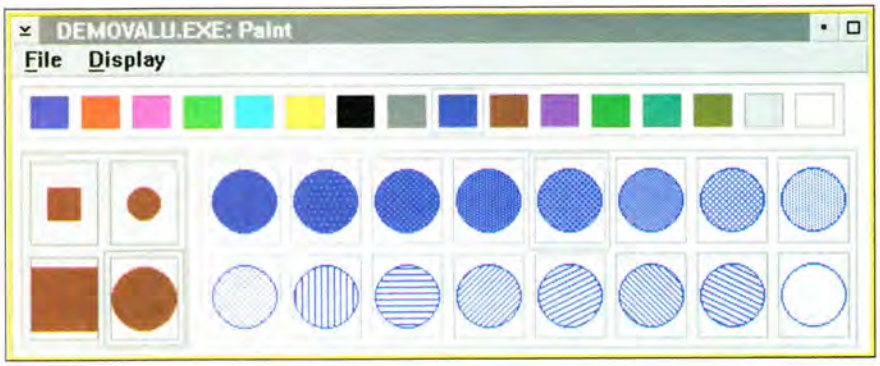


Figure 3. A Value Set Control

Slider

The Slider Control is similar to a scroll bar in its appearance but is used for a different purpose. Where a scroll bar is used to scroll data in a window, the Slider can be used to set, modify, and display specific values. For example, Sliders are particularly useful when immediate visual or audio feedback is needed. Sliders could be used to set the brightness, color, and decibel level in a multimedia application.

Sliders consist of an arm in a shaft containing a range of values. Buttons can also be specified at either end of the shaft. The buttons can be used to incrementally change the value of the slider, or the user can drag the arm to change its value.

An example of the Slider Control can be seen in Figure 4.

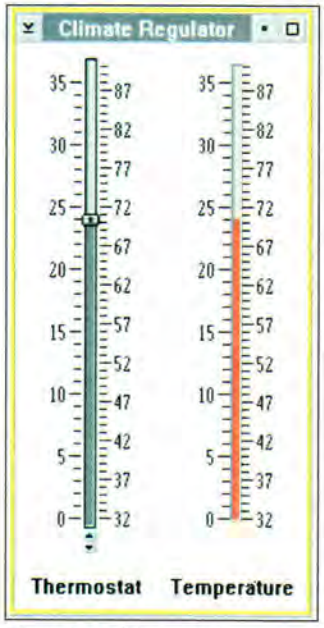


Figure 4. A slider control

Notebook

The Notebook Control simulates a real world notebook allowing the user to select and display pages quickly and easily.

The notebook provides binding, backward and forward page buttons, status line, major and minor tabs, and backpages to give a three dimensional appearance. The notebook can be set in different configurations depending on the needs of the application. For example, the binding can be placed on any one of the four sides to provide four visually unique, albeit similarly behaving, notebooks.

With this setup, the notebook can be used to organize like information in separate sections. Major tabs can denote breaks in sections, much like chapters in a book, while minor tabs can be used to further break down the sub-sections in a logical manner.

An example of the Notebook Control can be seen in Figure 5.

Figure 5. A Notebook Control

Container

The basic function of the Container Control is to hold objects. The objects held by the container are determined by the application. Information about these objects can be presented in a variety of views. Each view describes the objects in a different format, some giving different and/or additional information. The container supports the following views of its data:

Text View: Displays simple text lists. The items are displayed in one column, or multiple columns if the flowing option is specified.

Name View: Displays either icons or bit maps with text to the right. The items, like text view, are displayed in one column, or multiple columns if the flowing option is specified.

Icon View: Displays either icons or bit maps with text beneath them. Items may be displayed anywhere in the window.

Tree View: Displays a hierarchical list of items. The items can consist of only text, or, icons or bit maps with text to the right.

Details View: Displays detailed information about each item in the container with similar information arranged in columns. Details View supports icons, bit maps, text, and National Language Support (NLS) formatted dates and times. Details view also supports an optional split bar. The split bar can be dynamically placed between any two columns.

Each instance of text in the container can consist of unlimited lines and unlimited characters in each line. In addition, the container can hold as many objects as can be allocated by an application. The Container Control also supports the system drag protocol. This enables applications to drag container items within the current window, or transfer data from one container to another. The Container Control will become familiar to all users of OS/2 2.0 since it is being used extensively in the new Workplace Shell.

An example of the Container Control can be seen in Figure 6.

MORE BANG FOR YOUR BUCK

Why use the new PM Controls? The new PM Controls provide application developers with additional tools necessary to build consistent, powerful, CUA-compliant applications with less overhead. In addition, using these controls provides a means to develop a consistent user interface across a product platform. As new technology blossoms, the need for new controls becomes greater.

Consider a multimedia application that lets a user open and save files and set the volume, tone, brightness, and color of the display. In PM, there was no common control to open and save files. Hard to believe, isn't it? To set the volume, tone, etc., many applications would resort to using a scrollbar, but that is not how scrollbars were meant to be used. Lacking the proper PM controls, application developers had to write their own controls to get the function they needed. Without common controls, the operating sustem would have many applications written by different people, each with their own controls for opening and saving files, setting the volume, tone, etc.

With common controls as part of the operating system, all application developers will have the opportunity to use them. Therefore, the user interface across all applications that use these controls is the same. In addition, the code is very reusable. Application developers no longer have to reinvent a control each time they need it.

Figure 6. The Container Control in the Icon View

Peter Haggar, *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mr. Haggar is a senior associate programmer in OS/2 PM Extensions Development. He was a member of the Container control development team. He joined IBM in 1987 and has been working in OS/2 PM development since 1989. He received a BS in Computer Science from Clarkson University in NY.*

REFERENCES

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume II, (S10G-6265).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

Bernath, David. **File and Font Dialogs: Standardixzed Selection Techniques**, page 18, *IBM Personal Systems Developer*, Winter 1992.

Bernath, David, Jon Holliday. **Value Set Control: Selecting Graphical Information**, page 27, *IBM Personal Systems Developer*, Winter 1992.

Bernath, David, Jon Holliday. **Slider Control: Slip-Sliding Away in OS/2 2.0**, page 35, *IBM Personal Systems Developer*, Winter 1992.

Haggar, Peter, Tai Woo Nam and Ruth Anne Taylor. **Container Control: Implementing the Workplace Model**, page 48, *IBM Personal Systems Developer*, Winter 1992.

Mack, Diana. **Notebook Control: Organizing, Navigating, and Displaying Data**, page 44, *IBM Personal Systems Developer*, Winter 1992.

32

32-Bit OS/2 File and Font Dialogs: Standardized Selection Techniques

David Bernath

These two controls let a user select from available files and fonts

by David A. Bernath

An end goal of the Common User Access™ (CUA™) Architecture is to provide a consistent look and feel to the end user across applications. That is to say, knowledge learned by doing one task can be applied to other similar tasks resulting in a productivity gain. It is toward this end that the OS/2® standard File and Font Dialogs were written. These dialogs provide common, consistent functions which applications can use to help the end user increase his work.

Many applications work with information stored within files. Whenever the end user wishes to retrieve information stored within a file, he must select the name of the file from which to acquire the data. The same holds true when he wishes to save any work that is in progress for future reference. The methodology used today is that each application creates its own dialog to perform this task. The user must familiarize himself with these tasks and learn the dialog for each application.

OS/2 FILE DIALOG

The File Dialog control provides a dialog you can implement as either an Open or a SaveAs Dialog. The File Dialog control includes basic functions that you can extend to meet the requirements of your application. These basic functions give users the ability to:

- Display and select from a list of drives, directories, and files
- Enter a file name directly
- Filter the file names before they are displayed
- Specify extended attributes for TYPEs
- Interact with a single-selection or multiple-selection file dialog

Any application can easily use this system-provided function to save developers the overhead of writing the function and to give consistency across multiple applications.

Using the File Dialog

To present a basic file dialog to users, your application must take the following steps:

1) It must allocate a block of storage for a FILEDLG structure. This block is passed to the dialog from the application with the initial dialog settings.

2) Initialize the FILEDLG structure with the styles and initial settings you want. The control block size (cbSize) must be set to the size of the structure and the style flags parameter (fl) must have either the FDS_OPEN_DIALOG or the FDS_SAVEAS_DIALOG style bits set. The first denotes that the dialog will be an Open Dialog; used when an end user wishes the application to retrieve data from a file of his choosing, while the second means that the dialog will be of the SaveAs type, used when an end user wishes to save his data in a file also of his choosing.

3) Invoke the file dialog. Call WinFileDlg passing the dialog's parent and owner window as well as a pointer to the initialized FILEDLG structure.

This will prompt the end user for the information needed by the application. The final step left is to process the return value from the dialog. If the end user entered his selection, the application can perform the operation (Open, SaveAs) using the file name(s) returned in the dialog.

Customizing the File Dialog For Your Application

This is the File dialog in its simplest form. Fortunately, a wide range of functions have been provided to allow an application to customize the dialog to its choosing. Custom style flags can be set to provide these functions which include positioning of the dialog, both modal and modeless forms of the dialog, application specific tasks such as selection validation (including extended attribute support) and filtering support. These styles are outlined in Table 1. Additional customization is described below.

An application can also choose to initialize the FILEDLG structure with any values that users should see when they invoke the dialog for the first time.

A variety of styles are available for application customization

STYLE	DESCRIPTION
FDS_APPLYBUTTON	An Apply button is added to the dialog. This is useful in a modeless dialog.
FDS_CENTER	The dialog is positioned in the center of its owner window, overriding any specified x,y position.
FDS_CUSTOM	When this flag is set, a custom dialog template is used to create the dialog. The hMod and usDlgId parameters must be initialized.
FDS_ENABLEFILELB	When this flag is set, the file list box on a SaveAs dialog is enabled. When this flag is not set, the file list box is not enabled.
FDS_FILTERUNION	When this flag is set, the dialog uses the union of the string filter and the extended-attribute type filter when filtering files for the file list box. When this flag is not set, the list box, by default, uses the intersection of the two.
FDS_HELPBUTTON	A Help button is added to the dialog.
FDS_INCLUDE_EAS	When this flag is set, the dialog will always query extended attribute information for files as it fills the files listbox. The default is to not query the information unless an extended attribute type filter has been selected.
FDS_MODELESS	When this flag is set, the dialog is modeless.
FDS_MULTIPLESEL	When this flag is set, the file list box for the dialog is a multiple-selection list box. When this flag is not set, the default is a single-selection list box.
FDS_OPEN_DIALOG	When this flag is set, the dialog will be an OPEN dialog.
FDS_PRELOAD_VOLINFO	When this flag is set, the dialog will preload the volume information for the drives. The default behavior is for the volume label to be blank.
FDS_SAVEAS_DIALOG	When this flag is set, the dialog will be a SAVEAS dialog.

Table 1. File Dialog Style Flags

To initialize the drive field, your application should pass the name of the drive from which file information should first be displayed in the pszIDrive parameter. If you want to limit users' selections, pass a list of drives from which the user can choose in the papszIDriveList parameter. Otherwise, the system defaults to showing all available drives.

You can also pass the name of the initial file to be used by the dialog in the szFullfile parameter. This can be a file name or a string filter (for example *.DAT) to filter the initial file information in the file list box. This parameter can be fully qualified to select the initial drive and directory as well.

To initialize the type field, your application should pass the name of an extended-attribute type filter to be used to filter file information in the type parameter. Optionally, a list of extended attributes can be passed in the papszITypeList parameter. By selecting from this list, users can filter file information based on the extended attribute TYPE information.

An application can also choose to provide an application-specific title for the dialog. This might describe the type of action to be performed using the file. This can be done by passing a pointer to a null-terminated string in the pszTitle parameter. A user may also wish to use application-specific text for the OK button. If so, pass the pointer to a null-terminated string in the pszOkButton parameter to get this.

Users may also wish for a function which is not provided by the dialog. The application can easily add this function by specifying a custom dialog procedure. This procedure will provide the application-specific function desired. To do this, pass the pointer to a window procedure in the pfnwpDlgProc parameter to take advantage of this feature. A common example of this kind of function is adding a checkbox to the dialog to enable the end user to protect his files by making them read-only.

Additionally, an application can specify the position of the dialog by passing in the x,y position of the dialog. The dialog will be positioned there unless overridden by the FDS_CENTER flag, which will center the dialog on the owning application.

OS/2 FONT DIALOG

Another common action within an application is selecting a font with which to display or print text. The Font Dialog control enables users to view the font family facenames, styles, and sizes available in an application and select from them. The font family facename is defined as the name of the typeface. Courier, Times New Roman, and Helvetica are examples of commonly used family facenames. Type styles include normal, bold, italic, and bold italic. Size is defined as the point size, or vertical measurement of the type. Font emphasis styles include outline, underline, and strikeout. The Font Dialog control includes basic functions which you can extend to meet the requirements of your application. These basic functions give users the ability to:

- Display and select from a list of available font family names
- Display and select from a list of type styles for each font
- Display and select from a list of the available sizes for each font
- Display and select from a list of the emphasis styles available
- View their selections using a sample character string in a preview area

Using the Font Dialog

To present a basic font dialog to users, your application must take the following steps:

1) It must allocate a block of storage for a FONTDLG structure. This block is passed to the dialog from the application with the initial dialog settings.

STYLE	DESCRIPTION
FNTS_APPLYBUTTON	An Apply button is added to the dialog. This is useful in a modeless dialog.
FNTS_CENTER	The dialog is positioned in the center of its owner window, overriding any specified x,y position.
FNTS_CUSTOM	When this flag is set, a custom dialog template is used to create the dialog. The hMod and usDlgId parameters must be initialized.
FNTS_HELPBUTTON	A Help button is added to the dialog.
FNTS_MODELESS	When this flag is set, the dialog is modeless.
FNTS_OWNERDRAWPREVIEW	This flag makes the check boxes in the font dialog three-state check boxes and enables the user to leave certain style attributes "as is". The application is responsible for updating the preview area when this flag is set. This is useful when an application would like to have the user change an emphasis style for a document which has multiple fonts in it.
FNTS_RESETBUTTON	A Reset button is added to the dialog. When pressed, the initial values for the dialog will be restored from what the user has selected.
FNTS_BITMAPONLY	When this flag is set, the dialog will only present bitmap fonts. An application which changes fonts using the presentation parameters could make use of this flag.
FNTS_VECTORONLY	When this flag is set, the dialog will only present vector fonts.
FNTS_FIXEDWIDTHONLY	When this flag is set, the dialog will only present fixed width (monospaced) fonts.
FNTS_PROPORTIONALONLY	When this flag is set, the dialog will only present proportionally spaced fonts.
FNTS_NOSYNTHESIZEDFONTS	When this flag is set, the dialog will not synthesize any fonts.
FNTF_NOVIEWPRINTERFONTS	This flag is used when both the screen and printer presentation parameters are specified. If set, the font list will be filtered to not include any printer fonts.
FNTF_NOVIEWSCREENFONTS	This flag is used when both the screen and printer presentation parameters are specified. If set, the font list will be filtered to not include any screen fonts.

Table 2. Font Dialog Style Flags

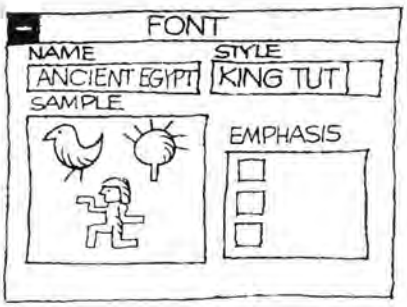
2) Initialize the FONTDLG structure with the styles and initial settings you want. The control block size (cb) must be set to the size of the structure and either the screen or printer presentation space parameter, or both, must be set. You must have a valid presentation space from which to query fonts.

3) Invoke the Font Dialog. Call WinFontDlg passing the dialog's parent and owner window as well as a pointer to the initialized FONTDLG structure.

This will prompt the end user for the information needed by the application to select a font. The final step left for the application is to process the returned information from the dialog. Lets say an application needs a font for displaying a set of text on the screen. It can present a font dialog to the end user. If the end user entered his selection, the application can easily change the font to the user's selection by using the information returned from the dialog.

Customizing the Font Dialog

This is the simplest form of the Font Dialog. There are many additional options provided to allow applications to perform a wide range of functions depending on its needs. Custom style flags can be set to allow positioning of the dialog, to select between a modal or modeless form of the dialog, to support default filtering of the font list, to allow ownerdraw capabilities in the preview window and more. These styles are outlined in Table 2. Additional customization is described in Table 2.



An application can also choose to initialize the FONTDLG structure with any values that users should see when they invoke the dialog for the first time.

To initialize the family facename, your application should pass that name in the pszFacename parameter. Otherwise, the dialog defaults to the first entry in the list of available family names.

To initialize the type styles, your application can set the weight class, width class and options for the facename selected. The weight class indicates the thickness of the strokes of the characters in the font. The width class indicates the relative aspect ratio of the characters of the font in relation to the normal aspect ratio for this type of font. Options indicate other characteristics about the font, for example; italic or rounded. To do this, your application should set the usWeight parameter to one of the FWEIGHT_* attributes, the usWidth parameter to one of the FWIDTH_* attributes, and the flType parameter to one of the FTYPE_* attributes. These are predefined system values which cover the spectrum of available choices. The dialog defaults to normal values for these if not specified.

To select the point size to use initially, your application should pass that size in the fxPointSize field. Otherwise, the dialog will default to the first available point size in the point size list. An application can also specify the point size list to be presented in the point size combobox. This can be done by passing in a null-terminated string containing point sizes separated by spaces in the pszPtSizeList parameter. A default list is provided containing point sizes 8, 10, 12, 14, 18, and 24.

To initialize the emphasis styles in effect, your application can set the flStyle field. The dialog defaults this to normal settings if not specified.

An application can also choose to provide an application-specific title for the dialog. This might describe the use for the font being selected. This can be done by passing a pointer to a null-terminated string in the pszTitle parameter. Additionally, the application can specify the text which appears in the preview area. Just pass the pointer to a null-terminated string in the pszPreview parameter.

If desired, an application can specify the position of the dialog by passing in the x,y position of the dialog. The dialog will be positioned there unless overridden by the FNTS_CENTER flag, which will center the dialog on the owning application.

An application may also wish for function which is not provided by the dialog. The application can easily add this function by specifying a custom dialog procedure. This

procedure will provide the application-specific function desired. Pass the pointer to a window procedure in the pfnwpDlgProc parameter to take advantage of this feature.

A common example of this kind of function is adding a control to allow color selection of the background and foreground colors of the font.

```
#define INCL_WINSTDFILE
extern CHAR szFileToBrowseUCCHMAXPATH~;

/*********************************************************************/
/* Procedure: GetFileToBrowse - This routine will prompt the user    */
/*            for the file to be browsed.  The user will select the  */
/*            fully qualified filename from the dialog.              */
/*                                                                   */
/* Inputs: hwndOwner - Owner handle for the dialog to prompt the     */
/*                     user.                                         */
/* Output: If TRUE, a file was selected                              */
/*********************************************************************/
BOOL GetFileToBrowse( HWND hwndOwner )
{
  FILEDLG fild;                 /* File dialog structure             */

  /*******************************************************************/
  /* Initialize the file dialog structure:                           */
  /*  The dialog should be an open dialog                            */
  /*  The dialog should be centered and provide a help button        */
  /*  The dialog should start with drive C as the initial drive      */
  /*******************************************************************/
  memset( fild, 0, sizeof(FILEDLG) );
  fild.cbSize    = sizeof(FILEDLG);
  fild.fl        = FDS_OPEN_DIALOG | FDS_CENTER | FDS_HELPBUTTON;
  fild.pszIDrive = "C:";

  /*******************************************************************/
  /* Bring up the File Dialog to prompt user for selection           */
  /*******************************************************************/
  WinFileDlg( HWND_DESKTOP, hwndOwner, &fild);

  /*******************************************************************/
  /* Check if the user selected a file or not.                       */
  /*******************************************************************/
  if (fild.lReturn == DID_OK )
  {
    /*****************************************************************/
    /* If a name was selected, copy the name to the file to browse   */
    /*****************************************************************/
    strcpy( szFileToBrowse, fild.szFullFile );
    return TRUE;
  }
  return FALSE;
}
```

Figure 1. Sample Code Demonstrating File Dialog Invocation

A SAMPLE PROGRAM

The sample program demonstrates the use of both of the dialogs. It allows a user to select a file to browse using the File Dialog and to select the font to display the contents of the file using the Font Dialog.

Selecting the Open pulldown from the File menu invokes the GetFileToBrowse routine (see Figure 1). This routine initializes a FILEDLG structure as an open dialog, centered in the application window with a help button and sets the initial drive for the dialog to C:

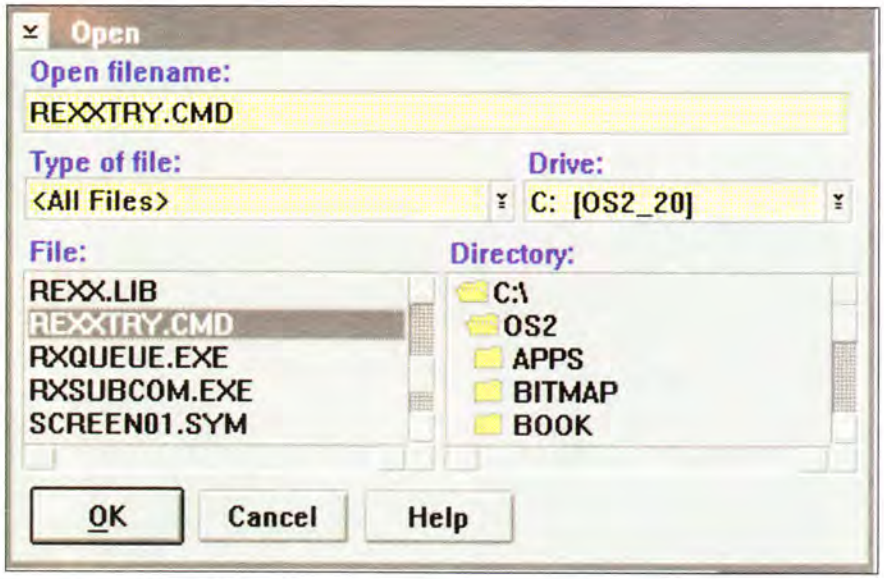


Figure 2. OS/2 Standard Open Dialog

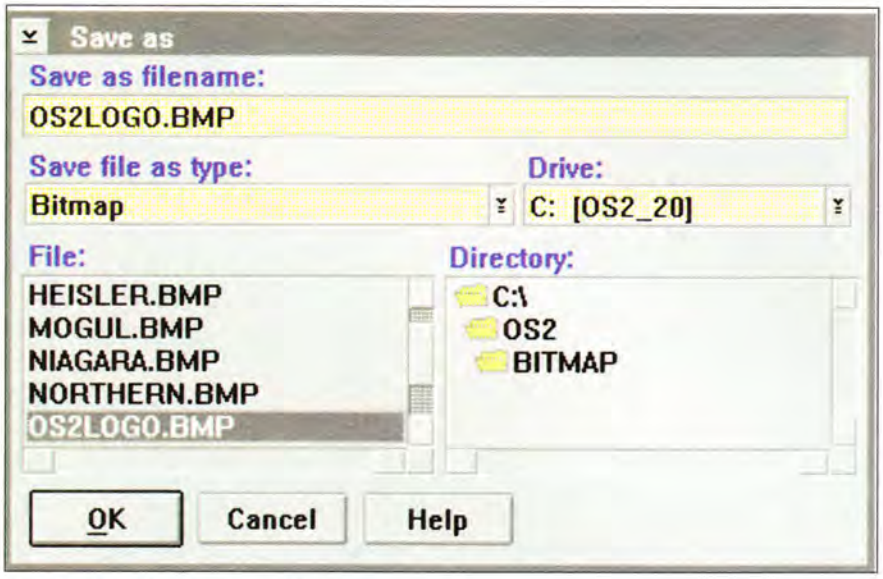


Figure 3. OS/2 Standard SaveAs Dialog

and then brings up the dialog (as pictured in Figure 2). When the dialog is dismissed, it checks the return information to see if a valid filename has been selected. If one has, it saves the filename in a global variable and returns TRUE to indicate a file was selected. An edit application might also contain a SaveAs option. Selecting this option could invoke the dialog as a SaveAs Dialog (as pictured in Figure 3).

Selecting the Change font pulldown from the Options menu invokes the SelectDisplayFont routine (see Figure 4). This routine initializes a FONTDLG structure centered in the application window with a help button. The preview area is to display the font with black text on a window colored background. It is to display screen fonts only; the initial font size is 8 point and the initial weight and width of the font are normal. The dialog is then brought up to allow the user to select from it (as seen in Figure 5). When the dialog is dismissed, the routine checks the return information to see if a font has been selected. If one has, it saves the font attribute structure returned (FATTRS) into a global variable for use in setting the font for the client window using GpiCreateLogFont. The routine also returns TRUE to indicate that the font has been updated.

These samples demonstrate how easy it is to use the dialogs to get the information that you need from your end user. Although a wide range of function is provided, the simplest applications can make use of these to provide a consistent look and feel to the end user with a minimal coding effort. These common, consistent functions will help both the application developer and end user gain productivity. Properly used, these dialogs can provide powerful function to your application without adding unnecessary code while making it compliant with CUA architecture.


```
#define INCL_WINSTDFONT
extern FATTRS fAttrs;

/*********************************************************************/
/* Procedure: SelectDisplayFont - This routine will change the font  */
/*            to be used to display the file being browsed           */
/*                                                                   */
/* Inputs: hwndOwner - Owner handle for the dialog to prompt the     */
/*                     user.                                         */
/* Output: If TRUE, a font was selected                              */
/*********************************************************************/
BOOL SelectDisplayFont( HWND hwndOwner )
{
  FONTDLG  Fntd;
  HWND     hWndFontDlg;
  CHAR     szFamilynameUFACESIZE~;

  /*******************************************************************/
  /* Initialize font dialog parameters for invocation                */
  /*  The dialog should be centered and provide a help button        */
  /*  The dialog should have black text on the window background     */
  /*  The initial size to be displayed should be 8 point             */
  /*  The initial weight and width attributes will be normal         */
  /*******************************************************************/
  memset( &Fntd, 0, sizeof(FONTDLG) );
  Fntd.cbSize = sizeof(FONTDLG);
  Fntd.fl = FNTS_HELPBUTTON | FNTS_CENTER;
  Fntd.clrFore = CLR_BLACK;
  Fntd.clrBack = SYSCLR_WINDOW;
  Fntd.fxPointSize = MAKEFIXED( 8, 0 );
  Fntd.hpsScreen = WinGetPS(hwndOwner);
  Fntd.usWeight = 5;
  Fntd.usWidth = 5;
  szFamilynameU0~ = '\0';
  Fntd.pszFamilyname = szFamilyname;
  Fntd.usFamilyBufLen = FACESIZE;

  /*******************************************************************/
  /* Prompt the end user to select a font for use in browsing        */
  /*******************************************************************/
  hWndFontDlg = WinFontDlg( HWND_DESKTOP, hwndOwner, &Fntd );

  /*******************************************************************/
  /* If one was selected, save font attribute structure returned for */
  /*  use in displaying the file and return TRUE since font selected */
  /*******************************************************************/
  WinReleasePS( Fntd.hpsScreen );
  if (( hWndFontDlg ) && ( Fntd.lReturn == DID_OK ))
  {
    memcpy( &fAttrs, &Fntd.fAttrs, sizeof(FATTRS) );
    return (TRUE);
  }
  return (FALSE);
}
```

Figure 4. Sample Code Demonstrating Font Dialog Interaction

REFERENCES

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume II, (S10G-6265).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

David A. Bernath, *2Bg/671/TB3, IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mr. Bernath is an advisory programmer in PM Extensions Development. He joined IBM in Boca Raton, FL in 1982, where he was involved in the development of the Generic Capture Test station for testing PCs and PS/2s during the manufacturing process. He is currently the technical lead for development of 32-bit Presentation Manager controls for OS/2 2.0. He received a BS in Computer Systems Engineering from Rensselaer Polytechnic Institute.*

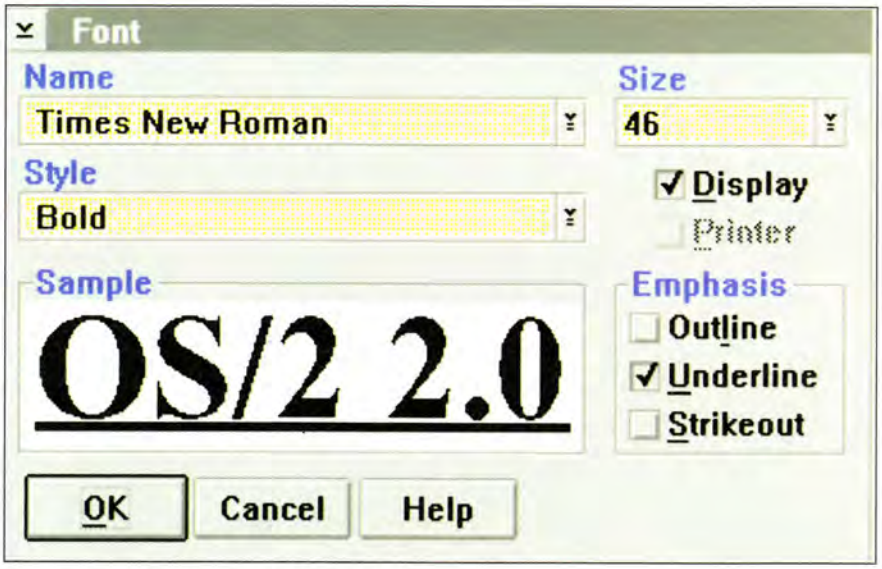


Figure 5. OS/2 Standard Font Selection Dialog

32-Bit OS/2

Value Set Control: Selecting Graphical Information

by David Bernath and Jon Holliday

OS/2® 2.0 introduces a new control for selecting graphical information from a set of choices. The Value Set control provides the capability to present graphical choices to the end user. Prior to OS/2 2.0, application developers had to write this function themselves since radio buttons supported only text.

The Value Set control is designed to be customizable to meet varying application requirements, while providing an easy-to-use interface component that can be used to develop products that conform to the Common User Access™ (CUA™) architecture guidelines. Value sets are typically used by applications when a color palette or tool palette would be presented to a user. The user would choose the color or tool desired from the set of choices presented. However, they can also be used to display small sets of numeric or textual information. Although radio buttons could also be used for this, in many instances using a Value Set can preserve space on the display screen.

Like radio buttons, a Value Set control is a visual component whose specific purpose is to allow a user to select one choice from a group of mutually exclusive choices. However, unlike radio buttons, a Value Set can use graphical images (bitmaps or icons), as well as colors, text, and numbers to represent the items that a user can select. A Value Set allows the user to see exactly what is being selected. This permits the user to make a selection faster than if the user had to read a description of each choice. For example, if you want to allow a user to select from a variety of patterns, you can present those patterns as Value Set choices, instead of having to provide a list of radio buttons with a description of each pattern.

A DEMO APPLICATION

An application can use this control within a window or dialog. To create it within a dialog, an application can use the Dialog Box Editor that is provided in the OS/2 2.0 Software Development Kit (SDK). This useful tool allows you to create WYSIWYG dialogs easily. For creating a Value Set control in a window, the following sample code and discussion illustrates the ease with which a Value Set can be used.

Figure 1. Sample Application Using Value Set Controls

David Bernath

Jon Holliday

The user can see exactly what is being selected

The sample application uses three Value Set controls to allow the end user to set values in a simple drawing program (see Figure 1). One of the Value Sets is used as a tool selector. This Value Set contains images (bitmaps) representing the tool to be selected. A second Value Set displays the colors which can be

```
/*********************************************************************/
/* Procedure: createValuesets - This procedure creates the value set  */
/*            controls used for the tool bar, color palette and font  */
/*            size selection in the demo drawing program              */
/*********************************************************************/
BOOL createValuesets( HWND hwnd )
{
  VSCDATA  vscData;
  ULONG    vsStyle;
  USHORT   idxRow;
  USHORT   idxCol;

  /*******************************************************************/
  /* Initialize the value set control block for tool bar selections  */
  /*******************************************************************/
  vscData.cbSize = sizeof(VSCDATA);
  vscData.usRowCount = 2;
  vscData.usColumnCount = 2;
  vsStyle = VS_BITMAP | VS_BORDER | WS_VISIBLE;

  /*******************************************************************/
  /* Create a value set control for the tool bar selections          */
  /*******************************************************************/
  hwndTool = WinCreateWindow( hwnd, WC_VALUESET, (PSZ)NULL,
             MAKELONG(vsStyle,0), 0, 0, 0, 0, hwnd,
             HWND_TOP, IDR_VSTOOLBAR, &vscData, (PVOID)NULL );

  /*******************************************************************/
  /* Now fill the tool bar with the icons for each tool to choose    */
  /*******************************************************************/
  for ( idxRow=1; idxRow <= vscData.usRowCount; idxRow++)
    for ( idxCol=1; idxCol <= vscData.usColumnCount; idxCol++)
    {
      /*****************************************************************/
      /* Place each tool bitmap into the proper row/column             */
      /*****************************************************************/
      WinSendMsg( hwndTool, VM_SETITEM, MPFROM2SHORT( idxRow,
                  idxCol ), MPFROMLONG( ahbmToolU (vscData.
                  usColumnCount * (idxRow-1)) + idxCol - 1 ));
    }

  /*******************************************************************/
  /* Initialize the value set control block for color palette        */
  /*******************************************************************/
  vscData.cbSize = sizeof(VSCDATA);
  vscData.usRowCount = 8;
```

Figure 2. Sample Code Demonstrating Value Set Creation (Continued)


```
vscData.usColumnCount = 2;
vsStyle = VS_COLORINDEX | VS_ITEMBORDER | VS_BORDER | WS_VISIBLE;

/*********************************************************************/
/* Create a value set control for the color palette selections       */
/*********************************************************************/
hwndColor = WinCreateWindow( hwnd, WC_VALUESET, (PSZ)NULL,
            MAKELONG(vsStyle,0), 0, 0, 0, 0, hwnd,
            HWND_TOP, IDR_VSPALETTE, &vscData, (PVOID)NULL );

/*********************************************************************/
/* Now fill the color palette with the colors to choose from         */
/*********************************************************************/
for ( idxCol=1; idxCol <= vscData.usColumnCount; idxCol++)
  for ( idxRow=1; idxRow <= vscData.usRowCount; idxRow++)
  {
    /*****************************************************************/
    /* Place the color indices from 1 to 16 into the cells           */
    /*****************************************************************/
    WinSendMsg( hwndColor, VM_SETITEM, MPFROM2SHORT( idxRow,
                idxCol ), MPFROMLONG( (vscData.usColumnCount *
                (idxRow-1)) + idxCol ));
  }

/*********************************************************************/
/* Initialize the value set control block for font selection         */
/*********************************************************************/
vscData.cbSize = sizeof(VSCDATA);
vscData.usRowCount = 1;
vscData.usColumnCount = 2;
vsStyle = VS_TEXT | VS_BORDER | WS_VISIBLE;

/*********************************************************************/
/* Create a value set control for the font selection                 */
/*********************************************************************/
hwndFont = WinCreateWindow( hwnd, WC_VALUESET, (PSZ)NULL,
           MAKELONG(vsStyle,0), 0, 0, 0, 0, hwnd,
           HWND_TOP, IDR_VSFONTS, &vscData, (PVOID)NULL );

/*********************************************************************/
/* Now fill the fontbox with the fonts to choose from                */
/*********************************************************************/
for ( idxCol=1; idxCol <= vscData.usColumnCount; idxCol++)
{
  /*******************************************************************/
  /* Place the text for the font name into each cell                 */
  /*  and disable each cell since text tool not selected             */
  /*******************************************************************/
  WinSendMsg( hwndFont, VM_SETITEM, MPFROM2SHORT( 1, idxCol ),
              MPFROMLONG( apszFontUidxCol-1~ ) );
  WinSendMsg( hwndFont, VM_SETITEMATTR, MPFROM2SHORT( 1, idxCol ),
              MPFROM2SHORT( VIA_DISABLED, TRUE ));
```

Figure 2. Sample Code Demonstrating Value Set Creation (Continued)

```
  }

  /*********************************************************************/
  /* Default tool selection to paint brush (Item in row 1, column 2)   */
  /*  and set default focus to the tool bar                            */
  /*********************************************************************/
  WinSendMsg( hwndTool, VM_SELECTITEM, MPFROM2SHORT( 1, 2 ), NULL );
  WinSetFocus( HWND_DESKTOP, hwndTool );
  return (FALSE);
}
```

Figure 2. Sample Code Demonstrating Value Set Creation

selected for drawing. The third Value Set contains point sizes which can be used for drawing text in the window.

CREATING A VALUE SET

The first thing you must do to create a Value Set control is to allocate and initialize a Value Set control data structure (VSCDATA). The VSCDATA structure contains the number of rows and columns of the Value Set you wish to create. The Value Set defaults to calculating the width and height of each item based on the size of the window specified and the number of rows and columns passed. In Figure 2, we are creating three Value Sets with varying row and column configurations. This routine is called within WM_CREATE processing of the

STYLE	DESCRIPTION
VS_BITMAP	The default attribute for each item in the value set will be set to VIA_BITMAP.
VS_ICON	The default attribute for each item in the value set will be set to VIA_ICON.
VS_TEXT	The default attribute for each item in the value set will be set to VIA_TEXT.
VS_RGB	The default attribute for each item in the value set will be set to VIA_RGB.
VS_COLORINDEX	The default attribute for each item in the value set will be set to VIA_COLORINDEX.
VS_BORDER	The value set will draw a thin border around itself to delineate the control.
VS_ITEMBORDER	The value set will draw a thin border around each item to delineate the items from each other.
VS_SCALEBITMAPS	The value set will scale bitmaps to the size of the cell containing the bitmap.
VS_RIGHTTOLEFT	The value set will interpret column orientation from right to left with the rightmost column being one (1) and counting up as you move left. Home will be interpreted as the rightmost column and End will be the leftmost column.

Table 1. Value Set Control Style Flags

sample applications window. The tool bar has 2 rows and 2 columns, the color palette has 8 rows and 2 columns and the font size selector has 1 row and 3 columns.

The next step is to create the Value Set window using the WinCreateWindow call. Your application should specify the size and position of the window; the styles to be used for the default type of item in each cell; any customization options for that ValueSet (see Table 1 for more details on styles); the parent and owner window handles; the window ID; and a pointer to the Value Set control data structure initialized above. The window class should be set to WC_VALUESET to specify that this is a Value Set window being created. Figure 2 illustrates three different styles of Value Sets being created, one containing bitmaps (the tool bar), one containing color indices (the color palette) and one containing text (the font size selector).

CUSTOMIZING AND INITIALIZING A VALUE SET

Once the Value Set window is created, the application should set the values of the items to be displayed within each cell of the Value Set. Additionally, the application can modify other Value Set attributes such as:

- modifying the size and width of each cell
- modifying the spacing between cells
- setting the initial selected cell
- changing the attributes of a Value Set item

In Figure 2, the items within each Value Set are filled with the type of item to be contained by them using the VM_SETITEM message. The tool bar Value Set is passed handles to previously loaded bitmaps which represent the tool at each item. The color palette is passed the index in the logical color table for the color to be represented at each item. The font size selector is passed a pointer to the string, which represents the point size, to be

STYLE	DESCRIPTION
VIA_BITMAP	This attribute is TRUE if the item is a bitmap. This is the default for a value set.
VIA_ICON	This attribute is TRUE if the item is an icon.
VIA_TEXT	This attribute is TRUE if the item is a text string.
VIA_RGB	This attribute is TRUE if the item is a color entry.
VIA_COLORINDEX	This attribute is TRUE if the item is an index into the logical color table.
VIA_OWNERDRAW	This attribute is TRUE if the application is to be notified whenever this item needs painting.
VIA_DISABLED	This attribute is TRUE if the item is disabled and cannot be selected.
VIA_DRAGGABLE	This attribute is TRUE if the item can be the source of a direct manipulation action.
VIA_DROPONABLE	This attribute is TRUE if the item can be the target of a direct manipulation action.

Table 2. Value Set Item Attributes

placed in each item. The Value Set interprets the information stored in each cell based on the item attributes of that cell and displays the information appropriately. The style flags used for the window when created determine the default attributes for each cell, but these may be changed by the application using the VM_SETITEMATTR message. Other attributes can be set for each item to specify certain characteristics and these are outlined in Table 2.

```
/*********************************************************************/
/* Process any control notifications here in WM_CONTROL              */
/*********************************************************************/
case WM_CONTROL:

  /*******************************************************************/
  /* Check what action occurred to cause this notification           */
  /*******************************************************************/
  switch (SHORT2FROMMP(mp1))
  {
    /*****************************************************************/
    /* If a select notification occurred, check which item from      */
    /*****************************************************************/
    case VN_SELECT:

      /***************************************************************/
      /* If tool bar item selected, check which tool was selected */
      /***************************************************************/
      if (SHORT1FROMMP(mp1) == IDR_VSTOOLBAR)
      {
        ulItem = (ULONG)WinSendMsg( hwndTool, VM_QUERYSELECTEDITEM,
                                    NULL, NULL );
        usToolNum = 2*(SHORT1FROMMR(ulItem)-1)+SHORT2FROMMR(ulItem);

        /*************************************************************/
        /* Enable font size selection if text tool selected          */
        /*************************************************************/
        for (idxCol=1; idxCol <= 2; idxCol++)
        {
          WinSendMsg( hwndFont, VM_SETITEMATTR, MPFROM2SHORT( 1,
          idxCol ), MPFROM2SHORT( VIA_DISABLED, (usToolNum != 3) ));
        }
      }
      /***************************************************************/
      /* Otherwise if color palette item selected                    */
      /***************************************************************/
      else if (SHORT1FROMMP(mp1) == IDR_VSPALETTE)
      {
        /*************************************************************/
        /* Check which color was just selected and get color value*/
        /*************************************************************/
        ulItem = (ULONG)WinSendMsg( hwndColor, VM_QUERYSELECTEDITEM,
```

Figure 3. Sample Code Demonstrating Value Set Interaction (Continued)

```
                                    NULL, NULL );
      ulColor = (ULONG)WinSendMsg( hwndColor, VM_QUERYITEM,
                                    MPFROMLONG(ulItem), NULL );
    }
    /*********************************************************/
    /* Otherwise if font size was selected (only enabled if the */
    /*  text tool is selected)                                */
    /*********************************************************/
    else if (SHORT1FROMMP(mp1) == IDR_VSFONTS)
    {
      VSTEXT vsText;

      /*******************************************************/
      /* Get which font item was selected ( column 1, 2 or 3 )  */
      /*******************************************************/
      ulItem = (ULONG)WinSendMsg( hwndFont, VM_QUERYSELECTEDITEM,
                                  NULL, NULL );
      vsText.pszItemText = szDrawFont;
      vsText.usBufLen = 14;

      /*******************************************************/
      /* Get the size that is saved as that item             */
      /*******************************************************/
      WinSendMsg( hwndFont, VM_QUERYITEM, MPFROMLONG(ulItem),
                  MPFROMP( &vsText ));

      /*******************************************************/
      /* Create a presparam string from the font size and name  */
      /* and set it for the drawing window                   */
      /*******************************************************/
      strcat( szDrawFont, szFont );
      WinSetPresParam( hwnd, (ULONG)PP_FONTNAMESIZE,
                    (ULONG)(strlen(szDrawFont) + 1), szDrawFont );
    }
    break;

  default:
    return WinDefWindowProc( hwnd, msg, mp1, mp2 );
}
break;
```

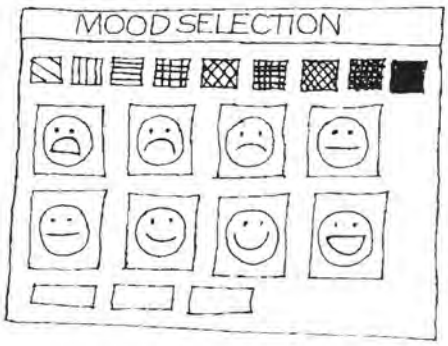
Figure 3. Sample Code Demonstrating Value Set Interaction

USING A VALUE SET

Once a Value Set is created, the application will be notified whenever the end user changes a selection. The VN_SELECT notification (via the WM_CONTROL message) will tell the application that a selection has changed for the indicated Value Set. The application can then query the Value Set for the new selected item using the VM_QUERYSELECTEDITEM message. Once the selected item is known, the application can then get the information stored at the selected item's location using the VM_QUERYITEM message. The sample application in Figure 3 shows the processing of the select notification message and the action taken by the application to determine which Value Set changed. Based on that, the application then takes the appropriate action to update itself based on the new setting, such as refreshing the color which is currently selected.

DIRECT MANIPULATION SUPPORT

The Value Set control also supports the drag-and-drop protocol within OS/2 2.0. This allows you to develop applications which can interoperate with other applications which support this protocol. For example, your application could allow an end user to dynamically configure a palette of selections by dragging items from a different application and dropping them within each cell. Although not shown in the sample, this powerful technique can be used to enhance productivity.



As a new control within OS/2 2.0, the Value Set control can help application developers create CUA architecture-compliant applications simply and reduce development time. In addition, use of a common graphical selection utility will help provide consistency between applications, requiring less training of end users.

REFERENCES

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

David A. Bernath, *2Bg/671/TB3, IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512-9968. Mr. Bernath is an advisory programmer in PM Extensions Development. He joined IBM in Boca Raton, FL in 1982, where he was involved in the development of the Generic Capture Test station for testing PCs and PS/2s during the manufacturing process. He is currently the technical lead for development of 32-bit Presentation Manager controls for OS/2 2.0. He received a BS in Computer Systems Engineering from Rensselaer Polytechnic Institute.*

Jon E. Holliday, *2Bg/671/TB3 IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512-9968. Mr. Holliday is a senior associate programmer in PM Extensions Development. He joined IBM in 1987 and worked on the OS/2 1.2 Dialog Manager. He is currently working on 32-bit PM Controls for OS/2 2.0, and "thunk" code to allow 16-bit and 32-bit programs to communicate. He received a BS in Computer Science from North Carolina State University.*

32-Bit OS/2

Slider Control: Slip-Sliding Away in OS/2 2.0

by David Bernath and Jon Holliday

Version 2.0 of IBM Operating System/2® introduces several new controls which make it easier for the application programmer to build Common User Access™ (CUA™) architecture-compliant applications. One of these controls is the Slider Control, which allows the user to set, display, or modify a value by moving a slider arm along the slider shaft. Prior to OS/2® 2.0, applications had to use the scroll bar control to obtain this function, which scroll bars were not intended to provide.

Sliders are typically used to allow a user to easily set values that have familiar increments, such as feet, inches, degrees, decibels, and so forth. However, they can also be used for other purposes when immediate visible feedback is necessary, such as to blend colors or to show the percentage of a task that has completed. For example, an application might allow a user to mix and match color shades by moving a slider arm, or a read-only slider could be provided that shows how much of a task has completed by filling in the slider shaft as the task progresses.

SLIDER COMPONENTS

First, let's discuss the different parts of a Slider Control (Figure 1). The slider arm shows the value that is currently selected by its position on the slider shaft. The slider arm can be moved along the shaft by clicking on the appropriate slider button, by clicking on the slider shaft, or by dragging the slider arm using the mouse. A tick mark is a mark that indicates an incremental value in a slider's scale. A detent is similar to a tick mark because it also represents a value on the scale. However, unlike a tick mark, a detent can be placed anywhere along the slider scale instead of in specific increments. The detent can then be selected causing the slider arm to move to the location of the detent.

A SAMPLE APPLICATION

An application can use this control within a window or dialog. To create it within a dialog, an application can use the Dialog Box Editor that is provided in the OS/2 2.0 Software Development Kit (SDK). This useful tool allows you to create WYSIWYG dialogs easily. Two Slider Controls created by the sample application are shown in Figure 2.

The sample application consists of two Slider Controls where one of the sliders represents a thermostat. The slider arm can be moved to set the desired temperature, which is represented in Fahrenheit on scale 1 and Celsius on scale 2. The second Slider Control represents a thermometer. This is a read-only Slider Control. It also shows Fahrenheit and Celsius degrees on its scales. The user cannot interact with the thermometer, or second, slider as this slider and ribbon strip are controlled by the application and display the temperature to which the thermostat slider is set.

CREATING A SLIDER

The first thing you must do to create a Slider Control is to allocate and initialize the Slider Control data structure (SLDCDATA). The SLDCDATA structure contains the scale information for the slider you wish to create. Depending on which scale(s) are used by your application, you must set the number of increments to be provided along each scale

David Bernath

Jon Holliday

The Slider Control gives the user immediate feedback

used. Optionally, the spacing between the increments can be set. If spacing is set to 0 as in Figure 3, the control calculates the spacing based on the Slider Control window size and the number of increments specified. In the example, the number of increments is set to 59

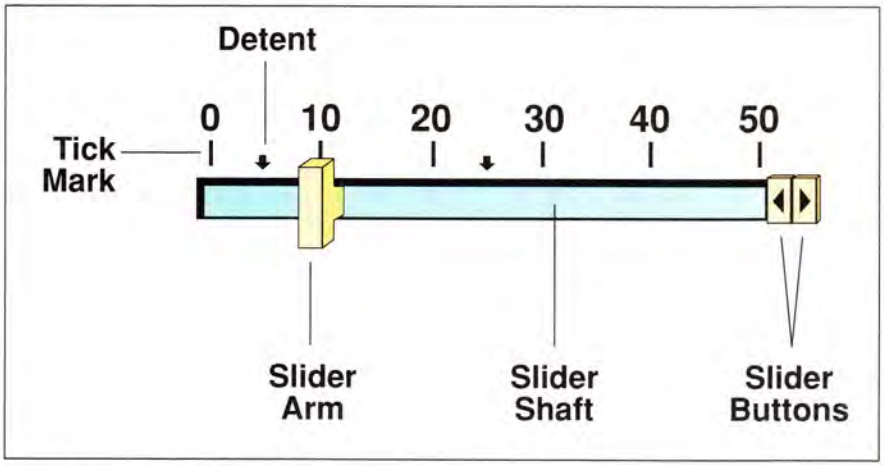


Figure 1. Parts of a Slider Control

for scale number 1 and 33 for scale number 2. Scale 2 is degrees Celsius and scale 1 is degrees Fahrenheit. By allowing the control to set the spacing of the increments, all the user needs to know is the range of Celsius degrees that covers the desired range of degrees Fahrenheit.

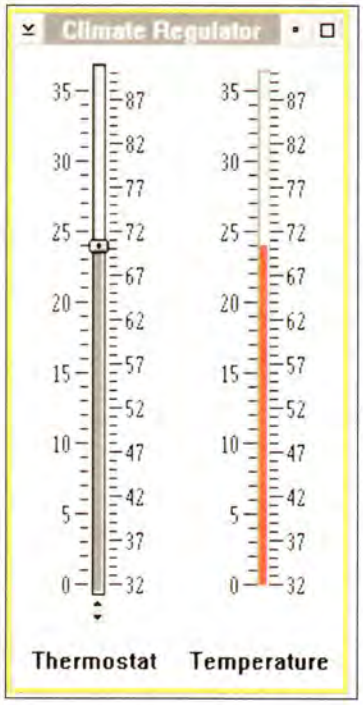


Figure 2. Slider Controls Representing a Thermometer and a Thermostat

Now we can create the Slider Control window using the Win-Create-Window call. Your application should specify the size and position of the window, the styles used for positioning, orientation, and customization of the slider within that window (see Table 1 for more details on styles), the parent and owner window handles, the window ID; and a pointer to the Slider Control data structure initialized above. The window class should be set to WC_SLIDER to specify that a slider window is being created.

STYLE	DESCRIPTION
SLS_HORIZONTAL	The slider will be presented in a horizontal orientation. The slider arm will move left and right on the slider shaft. Scales can be placed on the top and/or bottom of the slider. This is the default orientation of the slider.
SLS_VERTICAL	The slider will be presented in a vertical orientation. The slider arm will move up and down the slider shaft. Scales can be placed on the left and/or right of the slider.
SLS_CENTER	The slider will be centered in the slider window. This is the default positioning of the slider.
SLS_BOTTOM	The slider will be positioned at the bottom of the slider window (for horizontal sliders).
SLS_TOP	The slider will be positioned at the top of the slider window (for horizontal sliders).
SLS_LEFT	The slider will be positioned at the left edge of the slider window (for vertical sliders).

Table 1. Slider Control Style Flags (Continued)

STYLE	DESCRIPTION
SLS_RIGHT	The slider will be positioned at the right edge of the slider window (for vertical sliders).
SLS_SNAPTOINCREMENT	The slider arm position, when moved, will be adjusted to the nearest increment value (a rounding function), and will be redrawn at that position. If this style is not specified, the slider arm position remains at the spot it is moved to (no adjustment).
SLS_BUTTONSBOTTOM	The slider will include incremental adjustment buttons with the control and will place them at the bottom of the slider shaft (for vertical sliders). The buttons will increment the slider arm by one position.
SLS_BUTTONSTOP	The slider will include incremental adjustment buttons with the control and will place them at the top of the slider shaft (for vertical sliders). The buttons will increment the slider arm by one position.
SLS_BUTTONSLEFT	The slider will include incremental adjustment buttons with the control and will place them to the left of the slider shaft (for horizontal sliders). The buttons will increment the slider arm by one position.
SLS_BUTTONSRIGHT	The slider will include incremental adjustment buttons with the control and will place them to the right of the slider shaft (for horizontal sliders). The buttons will increment the slider arm by one position.
SLS_OWNERDRAW	The application is to be notified whenever the painting of the slider shaft, the slider arm, the ribbon strip, and the slider background is to take place.
SLS_READONLY	The slider is created as a read only slider. The user cannot interact with the slider. It is used merely as a mechanism to present a quantity to the user. Visual differences for a read only slider include a narrow slider arm, no buttons and no detents.
SLS_RIBBONSTRIP	The slider will fill the shaft on one side with a color value different from the slider shaft color as the arm is moved. The ribbon strip is between the home position and the slider arm and acts as water, filling a trough.
SLS_HOMEBOTTOM	The slider will use the bottom of the slider (for vertical sliders only) as the base value for incrementing. This is the default for vertical sliders.
SLS_HOMETOP	The slider will use the top of the slider (for vertical sliders only) as the base value for incrementing.

Table 1. Slider Control Style Flags (Continued)

STYLE	DESCRIPTION
SLS_HOMELEFT	The slider will use the left edge of the slider (for horizontal sliders only) as the base value for incrementing. This is the default for horizontal sliders.
SLS_HOMERIGHT	The slider will use the right edge of the slider (for horizontal sliders only) as the base value for incrementing.
SLS_PRIMARYSCALE1	The slider will use the increment and spacing specified for scale 1 as the incremental value for positioning the slider arm. Scale 1 will be displayed above the slider for horizontal sliders and to the right of the slider for vertical sliders. This is the default for a slider.
SLS_PRIMARYSCALE2	The slider will use the increment and spacing specified for scale 2 as the incremental value for positioning the slider arm. Scale 2 will be displayed below the slider for horizontal sliders and to the left of the slider for vertical sliders.

Table 1. Slider Control Style Flags

```
/*********************************************************************/
/*  Function:  createSliders                                         */
/*  Inputs:    hwnd - client window handle                           */
/*  Outputs:   none                                                  */
/*                                                                   */
/*  This function creates the temperature slider and the thermostat  */
/*  slider.                                                          */
/*********************************************************************/
BOOL createSliders (HWND hwnd)
{
   SLDCDATA        sldcData;          /* slider control data        */
   ULONG           wndStyle;          /* window style               */
   HWND         hwndTStatSld;         /* thermostat slider handle   */
   HWND         hwndTempSld;          /* temperature slider handle  */
   static char  szSldFont [20]  = "8.Tms Rmn";

   /*****************************************************************/
   /* Set up the thermostat slider                                  */
   /*****************************************************************/
   wndStyle = WS_VISIBLE        |   /* visible                     */
              SLS_BUTTONSBOTTOM |   /* buttons on the bottom       */
              SLS_RIBBONSTRIP   |   /* draw "progress indicator" strip*/
              SLS_VERTICAL      |   /* up and down                 */
              SLS_HOMEBOTTOM    |   /* count up from bottom        */
              SLS_SNAPTOINCREMENT;  /* slider cannot go to         */
                                    /* partial-increments          */
   /*****************************************************************/
   /* Initialize control block for thermostat slider                */
   /*****************************************************************/
   sldcData.cbSize             = sizeof (SLDCDATA);
   sldcData.usScale1Increments = 59;   /* 32 - 90 degrees Fahrenheit  */
```

Figure 3. Sample Code Demonstrating Slider Creation (Continued)


```
    sldcData.usScale1Spacing    = 0;    /* automatic increment spacing */
    sldcData.usScale2Increments = 33;   /* 0 - 32 degrees Celsius      */
    sldcData.usScale2Spacing    = 0;    /* automatic increment spacing */

    hwndTStatSld = WinCreateWindow( hwnd, WC_SLIDER, (PSZ) NULL,
                                    wndStyle, 0, YTOP_TEXT,
                                    CX_CLIENT / 2, CY_CLIENT - YTOP_TEXT,
                                    hwnd, HWND_TOP, IDR_TSTATSLIDE,
                                    (PVOID)&sldcData, (PVOID) NULL );

    /*******************************************************************/
    /* Change font for slider to something interesting                 */
    /*******************************************************************/
    WinSetPresParam (hwndTStatSld, (ULONG) PP_FONTNAMESIZE,
                     (ULONG) (strlen (szSldFont) + 1), szSldFont);

    /*******************************************************************/
    /* Initialize tick sizes and place text along scales               */
    /*******************************************************************/
    setupSliders(hwndTStatSld, &sldcData);

    /*******************************************************************/
    /* Set up the thermometer slider                                   */
    /*******************************************************************/
    wndStyle   = WS_VISIBLE         |    /* visible                    */
                 SLS_RIBBONSTRIP    |    /* progress indicator         */
                 SLS_VERTICAL       |    /* up and down                */
                 SLS_HOMEBOTTOM     |    /* count up from bottom       */
                 SLS_OWNERDRAW      |    /* ribbon will be drawn by app */
                 SLS_READONLY;           /* user cannot touch this     */

    /*******************************************************************/
    /* initialize control block for slider                             */
    /*******************************************************************/
    sldcData.cbSize             = sizeof (SLDCDATA);
    sldcData.usScale1Increments = 59;   /* 32 - 90 degrees Fahrenheit  */
    sldcData.usScale1Spacing    = 0;    /* automatic increment spacing */
    sldcData.usScale2Increments = 33;   /* 0 - 32 degrees Celsius      */
    sldcData.usScale2Spacing    = 0;    /* automatic increment spacing */

    hwndTempSld = WinCreateWindow ( hwnd, WC_SLIDER,
                                    (PSZ)NULL, wndStyle,
                                    CX_CLIENT / 2, YTOP_TEXT,
                                    CX_CLIENT / 2, CY_CLIENT - YTOP_TEXT,
                                    hwnd, HWND_TOP, IDR_TEMPSLIDE,
                                    (PVOID)&sldcData, (PVOID) NULL );

    /*******************************************************************/
    /* Change font for slider to something interesting                 */
    /*******************************************************************/
    WinSetPresParam (hwndTempSld, (ULONG) PP_FONTNAMESIZE,
                     (ULONG) (strlen (szSldFont) + 1), szSldFont);
```

Figure 3. Sample Code Demonstrating Slider Creation (Continued)

```
   /*********************************************************************/
   /* Initialize tick sizes and place text along scales of thermometer*/
   /*********************************************************************/
   setupSliders(hwndTempSld, &sldcData);
   return FALSE;
}
```

Figure 3. Sample Code Demonstrating Slider Creation

CUSTOMIZING A SLIDER

After the Slider Control window is created, but before it is made visible, the application can set other Slider Control characteristics, such as:

- the size and placement of tick marks
- text above/beside one or more tick marks
- one or more detents
- initial slider arm position
- modify slider shaft size
- modify slider arm size

In Figure 4, the sample code shows how to specify the size of the increments for each scale, and place text next to specific tick marks. If a detent was desired, this could be set at this time, before the slider is made visible. To change from scale 1 to scale 2, the application must change the primary scale to scale 2 by changing the style of the slider window. This can be done by calling WinSetWindowULong with an index of QWL_STYLE and ORing in the SLS_PRIMARYSCALE2 style.

```
/*********************************************************************/
/*  Function:  setupSliders                                          */
/*  Inputs:    hwnd - client window handle                           */
/*  Outputs:   none                                                  */
/*                                                                   */
/*  This function adds tick marks and text to a slider.              */
/*********************************************************************/
BOOL setupSliders (HWND hwndSlider, PSLDCDATA pSldcData)
{
   USHORT       idx;                  /* loop index                 */
   CHAR         buffer [5];           /* string buffer              */
   USHORT       usTickSize;           /* length of tick mark        */
   ULONG        ulSldStyle;           /* slider style               */

   /*********************************************************************/
   /* Place tick marks at the increments along slider scale 1           */
   /*********************************************************************/
   for ( idx = 0; idx <= (pSldcData->usScale1Increments); idx ++ )
   {
      /*****************************************************************/
      /* Tick size is 5 pels if not a multiple of 5, 8 pels otherwise */
      /*****************************************************************/
      usTickSize = 5 + ( (idx % 5) == 0) * 3 ;
      WinSendMsg ( hwndSlider, SLM_SETTICKSIZE,
                   MPFROM2SHORT ( idx, usTickSize), NULL );
   }
```

Figure 4. Sample Code Demonstrating Slider Customization (Continued)


```
   /*******************************************************************/
   /* Place text along scale 1 at desired increments (5 degrees)      */
   /*******************************************************************/
   for ( idx = 0; idx <= (pSldcData->usScale1Increments); idx += 5)
   {
      itoa (idx+32, buffer, 10);        /* convert number to string    */
      WinSendMsg ( hwndSlider,          /* and put into slider scale   */
                   SLM_SETSCALETEXT, MPFROMSHORT ( idx ),
                   MPFROMP ( buffer ) );
   }

   /*******************************************************************/
   /* Change scale to use to slider scale 2 to set up next tick marks */
   /*******************************************************************/
   ulSldStyle = WinQueryWindowULong( hwndSlider, QWL_STYLE );
   ulSldStyle |= SLS_PRIMARYSCALE2;
   WinSetWindowULong( hwndSlider, QWL_STYLE, ulSldStyle );

   /*******************************************************************/
   /* Place tick marks at the increments along slider scale 2         */
   /*******************************************************************/
   for ( idx = 0; idx <= (pSldcData->usScale2Increments); idx ++ )
   {
      /****************************************************************/
      /* tick size is 5 pels if not a multiple of 5, 8 pels otherwise */
      /****************************************************************/
      usTickSize = 5 + ( (idx % 5) == 0) * 3 ;
      WinSendMsg ( hwndSlider, SLM_SETTICKSIZE,
                   MPFROM2SHORT ( idx, usTickSize), NULL );
   }

   /*******************************************************************/
   /* Place text along ruler at selected increments ( 5 degrees )     */
   /*******************************************************************/
   for ( idx = 0; idx <= (pSldcData->usScale2Increments); idx += 5)
   {
      itoa (idx, buffer, 10);           /* convert number to string    */
      WinSendMsg ( hwndSlider,          /* and put into slider scale   */
                   SLM_SETSCALETEXT, MPFROMSHORT ( idx ),
                   MPFROMP ( buffer ) );
   }
   return TRUE;
}
```

Figure 4. Sample Code Demonstrating Slider Customization

SLIDER INTERACTION

Once a slider is created in your application, the user can interact with the slider to select the values which he desires. Your application can either dynamically monitor the changes as they are made, or retrieve them statically when the window is dismissed, such as when they are a part of a dialog.

The Slider Control provides two notification messages which allow your application to keep track of the position of the slider arm dynamically. The SLN_SLIDERTRACK

```
/*************************************************************/
/* The WM_CONTROL message is received to notify the application */
/*  that an event from one of its windows has occurred       */
/*************************************************************/
case WM_CONTROL:

  /***********************************************************/
  /* Check if this is a notification from the thermostat     */
  /***********************************************************/
  if ( SHORT1FROMMP(mp1) == IDR_TSTATSLIDE)
  {
    /*********************************************************/
    /* If so, check which notification was sent to the app   */
    /*********************************************************/
    switch (SHORT2FROMMP (mp1) )
    {
      case SLN_CHANGE:
      case SLN_SLIDERTRACK:
        /*****************************************************/
        /* If arm position has changed in the thermostat,    */
        /*  change the thermometer to the same reading       */
        /*****************************************************/
        WinSendMsg( hwndTemp, IDR_TEMPSLIDE ),
                    SLM_SETSLIDERINFO,
                    MPFROM2SHORT( SMA_SLIDERARMPOSITION,
                                  SMA_RANGEVALUE ), mp2 );
        break;

      default:
        break;
    }
  }
  break;

/*************************************************************/
/* The WM_DRAWITEM message is received if a slider was created  */
/*  with the ownerdraw style specified                       */
/*************************************************************/
case WM_DRAWITEM:
{
  /***********************************************************/
  /* If this is to draw the ribbon strip, we will draw it as a  */
  /*  red band (just like mercury in a thermometer)          */
  /***********************************************************/
  if (((POWNERITEM)mp2)->idItem == SDA_RIBBONSTRIP)
  {
    WinFillRect( ((POWNERITEM)mp2)->hps,
                 &((POWNERITEM)mp2)->rclItem, CLR_RED);
    return TRUE;
  }
  else
    return FALSE;
}
```

Figure 5. Sample Code Demonstrating Slider Notification Message Handling

notification (via a WM_CONTROL message) provides feedback as the slider arm is dragged but not yet released. The SLN_CHANGE notification tells an application that the slider arm position has changed, either via a keyboard or mouse action or by a message to set the slider's position. The sample application (Figure 5) uses these messages to keep track of where the user moves the slider arm on the thermostat Slider Control, so the ribbon strip of the temperature Slider Control can be painted to show the desired temperature. The temperature Slider Control was created with the SLS_OWNERDRAW style. Thus, the Slider Control sends a WM_DRAWITEM message to the application so the application can paint the desired parts of the slider.

An application can also retrieve the position of the slider statically by querying for it. Your application can do this by sending a SLM_QUERYSLIDERINFO message, passing as a parameter the SMA_SLIDERARM-POSITION. This position can be returned in one of two formats depending on an application's needs. If you are using the slider to select values that correspond to the increments on the slider, you would specify SMA_INCREMENTVALUE as an additional parameter to have the slider return the closest increment to the current position. If your application uses the slider to select an analog value from a range where the exact value is not as important, you would specify SMA_RANGEVALUE to have the slider return the position of the slider from home and the overall range of the slider.

To set the colors or font to be used in a Slider Control window, the application must send a WM_SETPRESPARAM message to the slider window, specifying a new color or font to use in the Slider Control window, as in Figure 3.

The Slider Control, one of many new controls in OS/2 2.0, allows application developers to create CUA architecture-compliant applications quickly and easily. The application can use the Slider Control to allow the user to set specific values as inputs, or the application can use the Slider Control as a progress indicator, providing feedback to the user on the progress of installations or other time consuming tasks.

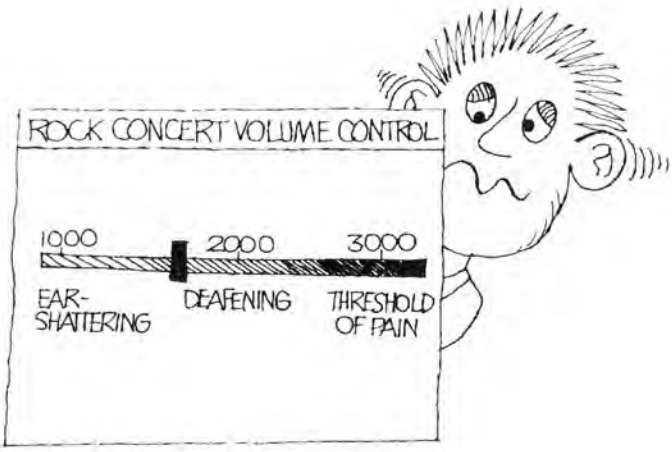
REFERENCES

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

David A. Bernath, *2Bg/671/TB3, IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512-9968. Mr. Bernath is an advisory programmer in PM Extensions Development. He joined IBM in Boca Raton, FL in 1982, where he was involved in the development of the Generic Capture Test station for testing PCs and PS/2s during the manufacturing process. He is currently the technical lead for development of 32-bit Presentation Manager controls for OS/2 2.0. He received a BS in Computer Systems Engineering from Rensselaer Polytechnic Institute.*

Jon E. Holliday, *2Bg/671/TB3, IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512-9968. Mr. Holliday is a senior associate programmer in PM Extensions Development. He joined IBM in 1987 and worked on the OS/2 1.2 Dialog Manager. He is currently working on 32-bit PM Controls for OS/2 2.0, and "thunk" code to allow 16-bit and 32-bit programs to communicate. He received a BS in Computer Science from North Carolina State University.*



32-Bit OS/2 Notebook Control: Organizing, Navigating, and Displaying Data

Diana Mack

This Control displays text in real-world Notebook fashion

by Diana Mack

In a continuing effort to provide a set of tools for the Workplace Model as defined by the SAA™ CUA Advanced Interface Design Guide, additional tools are needed to aid application developers in the implementation of this model. One such tool, the Notebook Control, simulates the real-world bound notebook paradigm, while providing a method for the organization, navigation, and display of data.

INTRODUCTION

The Notebook facilitates organizing related pieces of information into sections which the end user can easily find and view. The contents of each section relate to a common theme, which is represented by a divider similar to the tabbed dividers found in many loose-leaf notebooks. These dividers are organized in a manner consistent with the user's perception of the information contained within.

A possible use of the Notebook could be for the manipulation of data within the properties view of any standard object, while another possible implementation might be the display of the days of a month as depicted in a calendar.

Layout of the Notebook

The data in the Notebook is presented on pages and bound together to give an appearance similar to a real-world spiral bound notebook. Pages appear recessed on two edges of the book (i.e., backpages), thus providing a 3-D appearance. Like the real-world, the notebook is bound on one edge (binding). This provides an indication of how the pages will turn. At the intersection of the backpages are the page-turning buttons which allow the end user to navigate through the Notebook one page at a time. A forward page-turning arrow turns to the next page, while a backward page-turning arrow turns to the previous page. If desired, the user may also navigate directly from section to section.

Across from the Notebook binding are the major section dividers (major tabs). Orthogonal to the major tabs are the minor tabs which define sub-sections within the major sections. The section dividers provide a means of organizing the data within the book. Each section within the book may contain single or multiple pages. The notebook provides a method for the end user to turn the pages within a section and also to skip from one section to another easily. Just as dividers provide an indication of where the user is within the book, methods are supported to indicate where the user is within a section.

The visible area of the Notebook is the top page. The application uses this top page to display information and facilitate user interaction. The top page may contain application created windows or dialogs. Only one page is visible at any given time. The notebook handles the hiding and showing of the topmost window or dialog when pages are turned. Following are two examples showing how the Notebook can be used to to display a dialog and a window as the topmost page.

To illustrate thetop page implemented as a dialog, a properties notebook is used. Figure 1 shows how a properties notebook may look.

In this example, the various objects whose properties may be changed and/or updated are displayed as major tabs. Included are sections representing a folder, printer, and a display. The printer object is currently selected. Within the printer object the user may choose to "View" or "Update" the printer settings. These are represented as minor tabs within the printer object major tab section. The page is a printer dialog from which the user may update the printer name, type, and device information.

A calendar example is used to show how the top page may be implemented as a window. The calendar may be divided in various ways depending on the end user requirements. The calendar could include dividers for the months or weeks comprising a year. Additionally, this same data may be subdivided using two tiers of dividers, one for years and the other for months within the current year. This scenario provides a more compact view of the data while still providing a powerful navigation mechanism.

The calendar shown in Figure 2 is divided into four years (major tabs). Within each year are months (minor tabs) grouped in quarters. The major and minor tabs are presented on orthogonal sides allowing the end user to see all of the years and only those months for the currently selected major tab year. The top page is a window which shows the days for the currently selected month and year.

The Notebook Control was designed to be customizable to meet varying application requirements. The application may specify different colors, sizes, and orientations, however, the underlying function of the Control remains the same. As previously noted, the Notebook is bound on one edge, across from which are the major tabs. The major tabs represent the section divisions within the Notebook. Orthogonal to the major tabs are the minor tabs, which allow further division of the major tab sections. The minor tabs are placed based on the backpage orientation. The backpages provide the Notebook with the 3-D page affect. The minor tabs are shown only if the associated major tab is the topmost page.

An optional status text area is provided for the display of page-specific information. The status text provides an indication of where the page is, relative to the section in which it is contained (i.e., "page 1 of 4"). Additionally the status text may provide information about the page being displayed as in the printer dialog example (i.e., "Update Printer Setting"). The status text may be right justified, left justified, or centered on the notebook page.

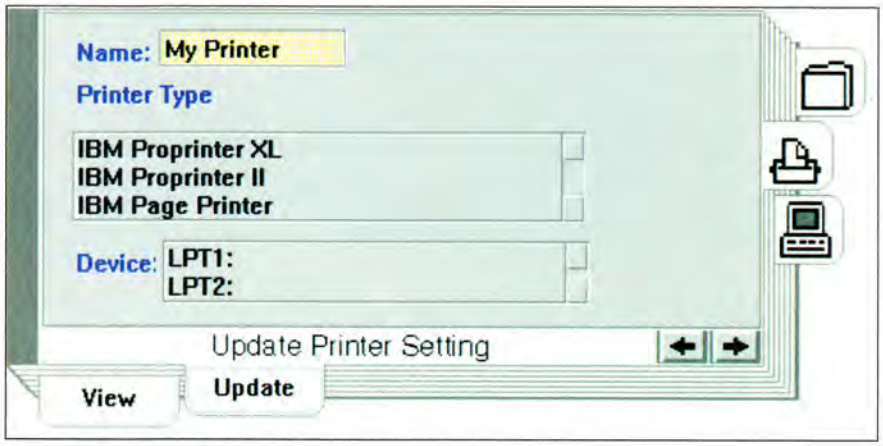


Figure 1. Dialog For Updating Printer Options

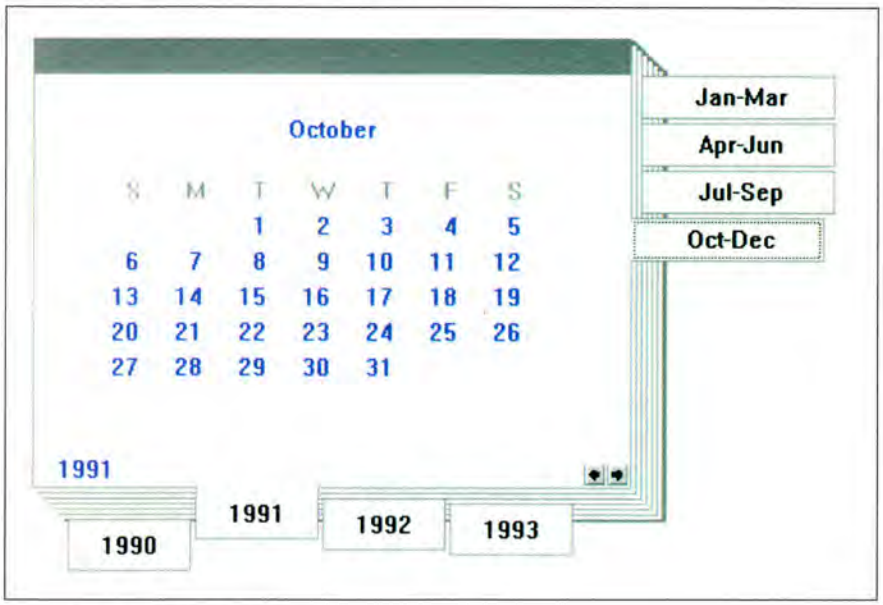


Figure 2. Notebook Calendar Example

Notebook as a Navigational or Organizational Tool

The Notebook control supports the use of a pointing device, such as a mouse, and the keyboard for displaying notebook pages and tabs, and for moving the selection cursor from the notebook tabs to the top page. The end user can turn from page to page or may go quickly from one tab page to another. Mnemonic key support is provided as a fast path for movement between the tab pages via single characters. If all of the tabs currently inserted cannot be displayed, scroll arrows are provided for scrolling the tabs forward and backward. Figure 3 shows a Notebook in which the scroll arrows are visible.

Figure 3. Customer Account Example

In the preceding example there is a major tab section which depicts the customer account folder. Within this section are the letters of the alphabet representing the first letter of the customer's last name. As shown, there are more letters than may be viewed on the book edge so scroll arrows are displayed to enable the viewing of those letters not shown. Cursor emphasis is provided to indicate which tab is currently selected.

The Notebook Control provides support for the dynamic insertion of pages. The insertion order of the pages determines the division of the sections (major tabs) and sub-sections (minor tabs). In the calendar example, if a page indicated by "1991" is inserted with a major attribute followed by the insertion of a page indicated by "Jan-Mar" with a minor attribute, the page indicated by "Jan-Mar" will be contained in the section delineated by "1991". If, however, the insertion order were reversed, the page indicated by "Jan-Mar" would not be contained in the section delineated by page "1991", but rather in the previous section "1990".

The contents of the Notebook can be changed by deleting pages, much as a user would tear pages out of the book. Support is provided to allow the deletion of a single page, a complete major or minor section, or all pages within the book. When pages are deleted, the division of the Notebook may change. For example, if a major tab page is deleted, any pages following that page, up to the next major tab page, will be included in the previous major tab section.

The Notebook allows the grouping of any combination of pages, major tab pages, and minor tab pages. Additionally, the application can define a book which contains no major or minor sections, but only pages.

Dynamic Resizing within the Notebook

The Notebook control is comprised of various regions which may be dynamically resized. Whenever the Notebook window is resized or any of the Notebook visual regions are resized, the Notebook dynamically recalculates the sizes of all affected regions for future display.

The application may dynamically set the size of the major tabs, minor tabs, and the page turning buttons. Once set, the Notebook determines the size of the top page which is bound by the Notebook backpages and binding. Support is provided for the application to determine the size of the page given the desired final Notebook size. The application may also determine the size of the Notebook given a top page size. The application will receive notifications when resizing is being performed.

Notebook Styles

The Notebook Control provides various window styles for tailoring the various visuals. The backpages which provide the Notebook with its 3-D appearance may intersect at any of the four edges of the book. Once the backpage intersection has been determined, the major tabs can be placed on one of the two edges from which the backpages intersect. The minor tabs will then be drawn on the the edge orthogonal to the major tab edge. The binding is drawn on the edge opposite the major tabs.

Three styles of tabs are available. Tab styles include square, round, and polygon. The tabs may contain text or bitmaps thus indicating the contents of the section. If the tab's contents are text, the text may be centered, left justified, or right justified when displayed.

CONCLUSION

As noted, the notebook is designed to be customizable to meet varying application requirements, while providing an easy-to-use user interface component that is consistent across multiple products. In this way, products can be developed that conform to the Common User Access™ (CUA™) user interface guidelines.

REFERENCES

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

Diana Mack, *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Ms. Mack is a staff programmer in OS/2 PM Extensions Development. She is the developer of the Notebook Control currently shipping in OS/2. She joined IBM in 1985 and has been working in OS/2 PM development since 1989. She received a BS in Computer Science from North Carolina State.*

32

32-Bit OS/2 Container Control: Implementing the Workplace Model

Peter Haggar

Tai Woo Nam

Ruth Anne Taylor

By Peter Haggar, Tai Woo Nam and Ruth Anne Taylor

As software development comes of age in the 1990s, the push is on to utilize reusable software components. This is particularly true in the area of the graphical user interface (GUI). As GUI evolves, the Common User Access™ (CUA™) architecture continues to define key components which provide consistent, usable interfaces. CUA's Workplace Model describes a component used to group related objects for easy access and retrieval and to present the objects in various views. This component is the Container Control, developed by the IBM PM Extensions group in Cary. The Workplace Model is embodied by the OS/2® 2.0 Shell, of which the Container Control is a key component of desktop management. The Container Control is part of the OS/2 2.0 base operating system application program interface (API).

This article defines, describes the major functions, and suggests uses of the Container Control. Tips for optimizing applications using the Container Control are provided.

CONTAINER CONTROL DEFINITION

The Container Control provides a framework for the application to display data. The data can be represented as objects and can be presented in different views. The Container provides a powerful, completely flexible, and easy-to-use user interface component for developing products that conform to the CUA interface guidelines. Objects within the Container Control are as varied as your applications. For example, one application may use the Container to represent a file system and its objects would be files and directories. Another application may show the contents of a database using a Container with the key elements as Container objects. Products often provide product-specific Containers that have special features to serve the needs of the product's users. A graphics product might provide a portfolio in which a user could store and sort artwork according to the subject matter or technique.

The Container offers the application developer the option of displaying data in the following views: Icon, Details, Name, Text, and Tree.

Icon View

The icon view, Figure 1, represents CUA's Workplace Model "messy desk" paradigm. The Container objects are presented as icons or bitmaps with text beneath.

If your application requires that the user be able to position the objects where desired by dragging and dropping them within the Container or from other sources, the icon view would be appropriate. The user may group the objects as preferred by overlapping them, or in "messy desktop" fashion. The icon view is not gridded; it has free-form characteristics. That is, data can be placed in various relative positions and still have meaning. If a gridded display format is preferred, the objects can be automatically positioned by using the autoposition function of Container. When the Container's autoposition style bit is set, the items displayed in icon view are arranged in rows from left to right and from top to bottom.

Direct manipulation is a protocol that allows the user to drag Container items within a current window or from one window to another. While dragging within the same

Container window the user can reposition or reorder Container items. Direct manipulation is supported in all views of the Container Control.

The OS/2 2.0 Workplace Shell uses a Container Control with a default icon view for desktop management. Users may place applications and folders represented as icons with text strings on the desktop. The desktop may be arranged as the user desires, and whenever it is closed, it is restored to the user's configuration when opened.

Details View

The details view, Figure 2, provides a means of displaying detailed information about items in an unlimited number of scrollable columns with optional column headings. The data within each column may be icons, bitmaps, text strings, or National Language Support (NLS) formatted date or time strings. An optional splitbar may be used to divide the display area into two windows which scroll independently horizontally and dependently vertically.

If your application requires that data be presented in a tabular format or that supplementary information be provided for each data item, the details view is suitable. File management or database applications are particularly well suited to the details view. The contents and details views described by CUA in the "SAA CUA Guide to User Interface Design" can be implemented with the Container's details view (see reference box at end of article).

Name View

The name view, Figure 3, displays Container objects as icons or bitmaps with the text strings to the right in columnar format. Optionally, the columns may be flowed to fit the window as it is sized. If necessary, scroll bars are provided, in this and all other views, so that all data may be displayed. If your application requires that the objects be represented by icons or bitmaps and that the data remains in columns, the name view is appropriate. An application which uses icon/text pairs arranged in columns for user selection, such as a touch-screen menu, would be ideally suited for name view display.

Text View

The text view, Figure 4, is similar to the Presentation Manager listbox in OS/2. Text strings are displayed in columnar format. Similar to the name view, the text view also offers the option of flowing the text into multiple columns to fit the window when it is

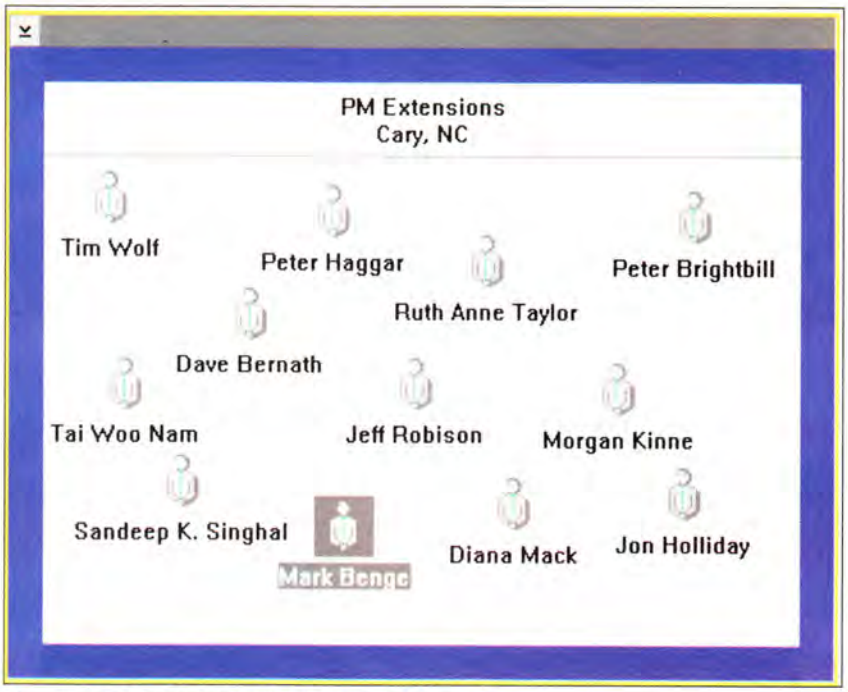


Figure 1. CUA's Workplace Model "messy desk" paradigm

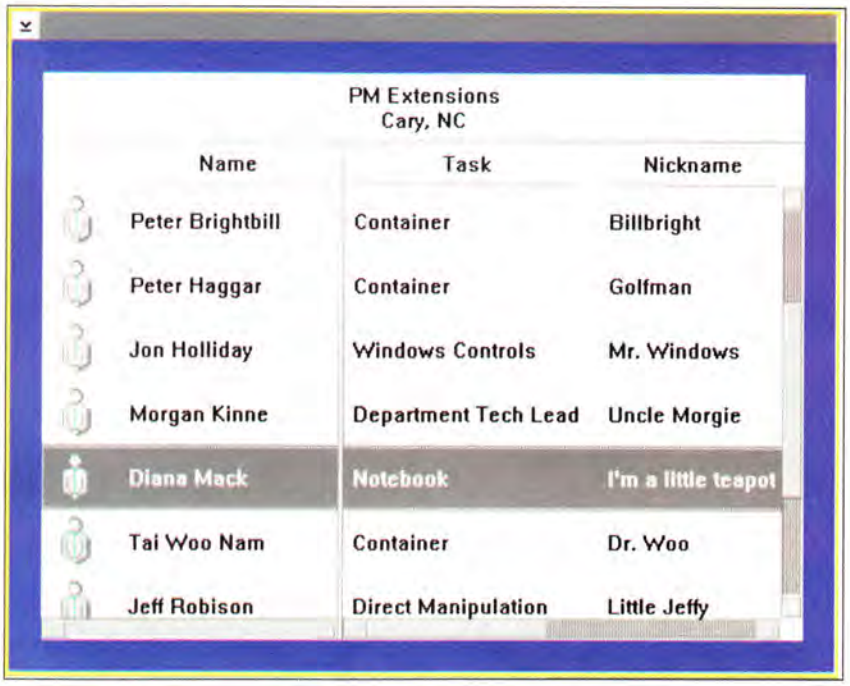


Figure 2. The Details View Displays Detailed Information About Items in an Unlimited Number of Scrollable Columns With Optional Column Headings

sized. Applications which want the format of the system listbox control and would like the additional functionality of the Container Control will benefit from the text view. (Refer to "Major Function of the Container Control" for a description of the Container Control functionality.) An application which needs a bridge to a text-based application can benefit from the text view.

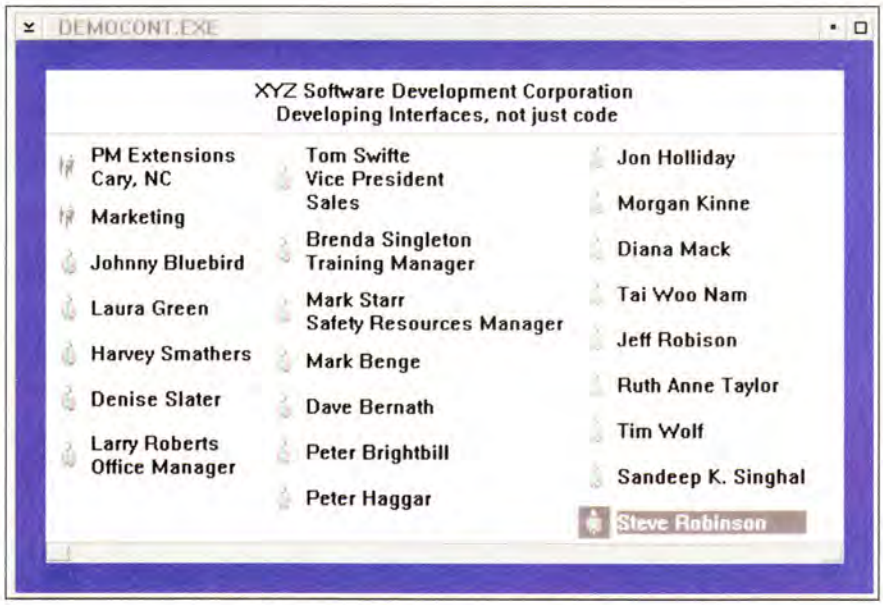


Figure 3. Name View Displays Container Objects as Icons or Bitmaps With the Text Strings to the right in Columnar Format

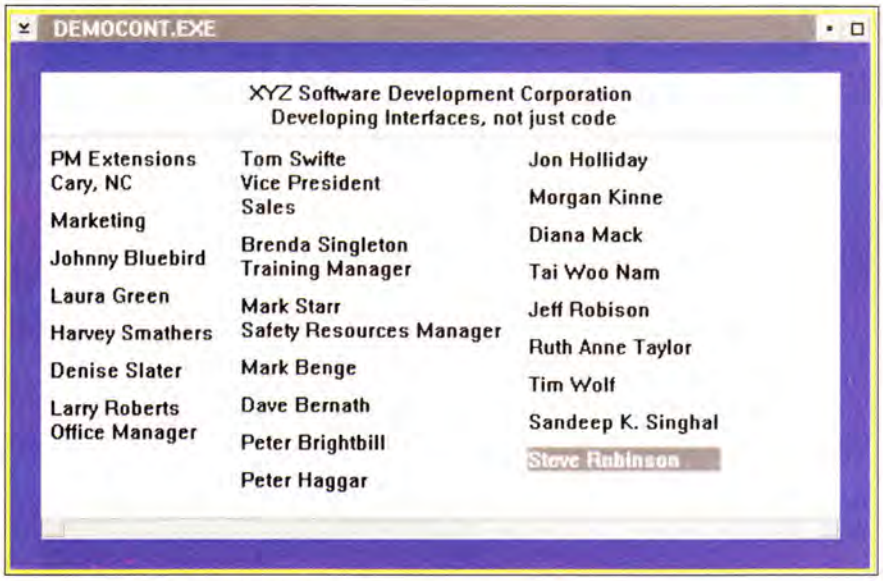


Figure 4. Text Strings are Displayed in Columnar Format

Tree View

The tree view, Figure 5, presents data in a hierarchical format. The leftmost items displayed in the tree view are at the root level and are the same items that are displayed in all of the other Container views. Root level items can contain other items called children and which can be displayed in the tree view. If the children are not displayed, the parent item can be expanded to display them as a new branch in the tree view. Once a parent item has been expanded, it can then be collapsed to remove its children from the display.

An application such as a file system, which is represented in a data hierarchy, can be presented with the tree view. The OS/2 2.0 Workplace Shell uses the tree view to display the drives, directories, and files on your computer.

MAJOR FUNCTION OF THE CONTAINER CONTROL

The Container Control provides a rich set of functions for application developers and end users. Base function of the Container includes no specific limit on the number of items in the Container and unlimited text. All text strings displayed in the Container can have unlimited lines; each line may have unlimited characters. This allows applications the flexibility to display as much text as they wish, without reaching a text limit or truncating data. The number of items in the Container is only limited by the amount of memory on a user's machine.

Container Appearance

There is great flexibility in altering the appearance of the Container Control and of individual items. The application developer may customize the Container background or individual Container items through ownerdraw support. Ownerdraw is also supported on a per column basis in details view so that individual columns may be drawn.

The font and color can be changed for the text in all views. Fonts and colors can be set and changed dynamically using the PM WinSetPresParam function call.

The Container Control supports the drawing of icons or bitmaps in each view. Similar existing controls do not support the display of both icons and bitmaps, limiting the developer's flexibility. Icons and bitmaps can also be used as column titles in details view.

User Interaction

The end user can alter any text field in the Container including the Container title, column headings in details view and all Container items through direct editing. The user is allowed to change all of these text strings at any time. If there are text strings the application does not want the user to change, they can be individually set to read-only.

Support is also provided for the user to directly edit a blank text string. For example, text may not be specified for a particular item or title. In this case the Container allows the user to create a text string for this blank field via direct edit.

Not only can the user interact with the Container through direct editing and direct manipulation, but objects may be selected through marquee, touch swipe, range swipe, and first letter selection techniques. The Container Control supports single, extended, and multiple selection types specified in the *SAA CUA Advanced Interface Design Reference*.

Data Manipulation

Application developers have flexibility in the way data can be structured. For example, if items should be visible at some times and hidden at other times, filtering may be used to hide items from view. The Container determines the viewable set of items via an application-defined filter function. Filtered items can be unfiltered to make them visible again.

Similarly, columns in details view may need to be hidden and unhidden dynamically. The Container provides the application the ability to set each column as visible or invisible. Making a column invisible is an easy way to remove a column from view without having to actually remove it from the Container.

The application developer may order the Container items via an application-defined sort function. Sorting can be done as the items are inserted into the Container or after insertion is complete.

TIPS FOR ENHANCING PERFORMANCE OF THE CONTAINER CONTROL

There are many ways application developers can enhance the performance of their application using the Container Control. We have included some of the more common tips developers can use when writing applications using the Container Control.

Figure 5. Presents Data in a Hierarchical Format

Record Insertion and Removal

Inserting and removing records into and from the Container can be done one by one or in a block. Block insertions and removals refer to processing more than one record at a time. The block processing method is preferred not only because message traffic is reduced, but also because the amount of code executed inside of the Container is minimized.

The Container Control gives function and flexibility to the GUI developer

If your application does not allow you to process insertion and removal using blocks, performance can still be enhanced using the flags provided with the CM_INSERTRECORD and CM_REMOVERECORD messages. A flag is provided in each message to tell the Container whether to invalidate and update the Container after a record has been inserted or removed. If you do not wish for each record to be invalidated upon insertion and removal, this flag should be set off for all records inserted or removed individually. After all records are processed, the application should send the CM_INVALIDATERECORD message to refresh the entire Container. This method will greatly enhanced Container performance .

Performance Considerations for Icons and Bitmaps

There are certain characteristics inherent to both icons and bitmaps that application developers may wish to consider before deciding which to use.

The application can use both icons and bitmaps to display objects in the Container Control. In addition, the size of the icons and bitmaps can be specified by the application. The overall performance of the Container is best when the system default size is used for all icons or bitmaps. This size corresponds to the value returned from the WinQuerySysValue call specifying SV_CXICON and SV_CYICON. When the system default size is used, the Container does not have to do any compressing or stretching which enhances performance as well as the resolution of the icon or bitmap.

The background of an icon in the Container is transparent. This allows for a nicer appearance in icon view when the icons are stacked on top of each other and when the icon is being dragged. Unlike icons, the background of a bitmap is not transparent. Emphasis states, such as selection, will be drawn around the edge of bitmap. If the bitmap image does not fill the bitmap rectangle, the bitmap will appear as the image surrounded by a background rectangle and bordered by the emphasis state. Icons display the emphasis states differently. If the image does not fill the icon rectangle, the emphasis state fills the icon rectangle.

An icon is composed of two bitmaps. If the application has specified that the Container size an icon, the Container compresses or stretches two bitmaps for each icon, resulting in increased execution time. If the compressing or stretching is done at the time the icon is drawn, performance would be unacceptable. To overcome this, the two sized bitmaps are stored in an internal icon metrics at the time the item is inserted. Therefore, record insertion performance is decreased when icons need to be sized.

If the application specifies the bitmap to be sized, the Container does not save the compressed or stretched bitmap. Instead, compressing or stretching is done at the time the bitmap is drawn. Therefore, inserting records with bitmaps is faster than icons when sizing is needed, but drawing time is slower.

When the bitmaps for sized icons are saved in the internal icon metrics, each icon with a different handle (HPOINTER) is stored separately. Therefore, when using the same icon for different records, the same icon handle (HPOINTER) should be used to optimize performance. This means the application should only load the icon once using WinLoadPointer, and use the returned handle repeatedly. The advantages include increased performance due to reducing the number of icons to be loaded and sized and a reduction in the memory required for the internal metrics.

Painting optimization using CM_INVALIDATERECORD

The invalidation of a record is a time consuming process because of the characteristics of unlimited items, unlimited lines of text for each item and unlimited characters in a line. To avoid unnecessary processing the Container provides four flags for use with the CM_INVALIDATERECORD message. These flags are:

CMA_ERASE: This flag is used only for icon view when the application is displaying icons.

Since the background of an icon in the Container is transparent, changing the icon to have a smaller image will not completely erase the previous larger image. See Figure 6. CMA_ERASE should be specified for proper invalidation.

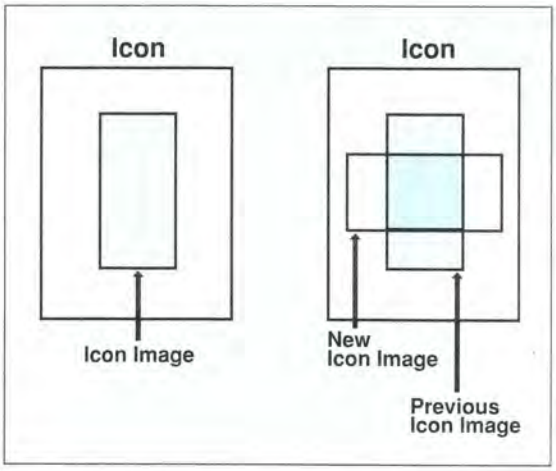


Figure 6. Icon Images

Stacked icons are redrawn from the bottom to the top. Therefore, if the CMA_ERASE flag is not specified, those icons on the bottom are not completely redrawn when the size of the top icon image changes. Refer to Figure 7.

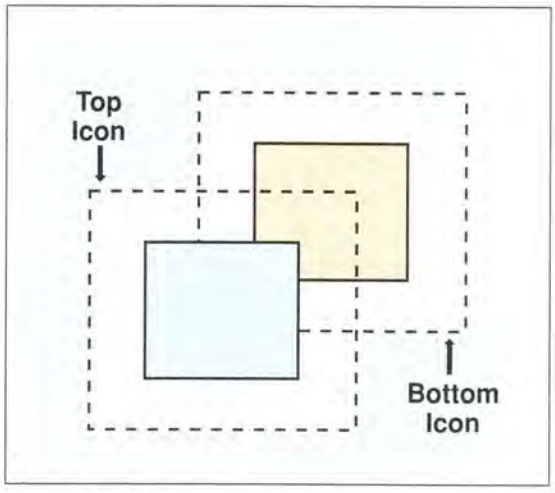


Figure 7. Stacked Icons

CMA_REPOSITION: All items will be repositioned and the viewport will be repainted when this flag is specified. Use of the reposition flag results in the greatest performance degradation as it causes the most repainting and repositioning of any invalidation flags.

CMA_NOREPOSITION: This flag produces the most optimized results as it will redraw only the items specified. The Container will not do any positioning or recalculations. The application developer should only use this flag if an item's icon or text changes and the height and length of the item are the same.

CMA_TEXTCHANGED: When the text of an item changes and the application is unsure of its effect on other items in the Container, the CMA_TEXTCHANGED flag should be used. In this case, the Container will determine if any repositioning of items is needed. For optimal performance, only those items which require positioning and painting will be refreshed.

SUMMARY

The Container Control offers great flexibility and rich functionality for the developer who is writing applications with a graphical user interface. The use of the Container Control can be tailored to fit the application and its data display requirements. The availability of the Container Control as a reusable component in the OS/2 2.0 operating system makes this possible.

REFERENCES

SAA CUA Guide to User Interface Design, (SC34-4289).

SAA CUA Advanced Interface Design Reference, (SC34-4290).

OS/2 2.0 Programmers Guide Volume II, (S10G-6494).

OS/2 2.0 Presentation Manager Programming Reference Volume III, (S10G-6272).

32

Peter Haggar, *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mr. Haggar is a senior associate programmer in OS/2 PM Extensions Development. He was a member of the Container Control development team. He joined IBM in 1987 and has been working in OS/2 PM development since 1989. He received a BS in Computer Science from Clarkson University, NY.*

Tai Woo Nam, *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mr. Nam is an associate programmer in OS/2 PM Extensions Development. He was a member of the Container Control development team. He joined IBM in 1989 and has been working in OS/2 PM development since then. He received a BS in Computer Science, Economics, and Biochemistry from Rutgers University, NJ.*

Ruth Anne Taylor, *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mrs. Taylor is a senior associate programmer in OS/2 PM Extensions Development. She was a member of the Container Control development team. She joined IBM in 1988 and has been working in OS/2 PM development since 1989. She received a BA from the University of Kentucky, and a BS in Engineering Mathematics and Computer Science, and a Master's of Engineering in Computer Science, both from the University of Louisville, KY.*

The IBM Cary PM Extensions Development Team, standing on bridge (l-r) Jeff Robison, Tim Wolf, Steve Robinson, Jon Holliday, Tai Woo Nam, Ruth Anne Taylor, David Bernath, Pete Brightbill, Diana Mack, Mark Benge, and Peter Haggar.

Software Tools

One-Stop Shopping for Compilers

by Brian Proffit

This issue's Tools Update column explores an integrated family of language products.

Religion, politics, text editors, and development languages; all things to avoid in conversation if you want things to remain polite. The C people think the assembler people are wasting development time writing code that's just as fast (so they say) when written in C. The C++ fans think that putting object-oriented extensions on other languages is a waste of time, while the Pascal and Modula-2 people think anything else is a waste of time. The COBOL people think all programmers ought to be paid by the word, but the APL folk say COBOL is Greek to them.

The simple fact is, of course, that each language is appropriate for certain kinds of tasks. While there have been several attempts to design the one complete Language — (remember PL/I?) — we're still subject to the limitations of whatever language we select, or are we?

Jensen & Partners International, Inc. (JPI) has created a family of languages that share a common development environment. Any language can call routines written in any other. If you're writing a Pascal program and missing C's *getenv* function to read an environment variable, why not write a small C function and call it from your Pascal program? The TopSpeed® family currently includes C, C++, Pascal, and Modula-2. The TopSpeed TechKit adds an assembler. Pick the language with which you're most comfortable and exploit the strengths of the others when it's beneficial to do so.

The glue that holds this all together is the TopSpeed Environment. New languages are easily plugged into the Environment giving the developer a consistent look-and-feel across all compilers.

The multiple-window editor allows up to nine open files at a time and supports cut/paste across the windows. There are good search and replace capabilities. And, if you have a problem, the context-sensitive help system is easily accessed. The editor includes a realtime syntax checker for Pascal and Modula-2 programs. The editor defaults to Wordstar™ command sequences, but that's easily customized. There is also a browse mode to avoid accidental updates. Compile errors are highlighted and the cursor is moved to the location of the first error, as you would expect from an integrated editor. The F8 key moves the cursor to subsequent errors, and the corresponding diagnostic is displayed.

The TopSpeed languages compile to a common pseudocode. The environment contains an optimizing code generator which converts the pseudocode to executable code. Type-safe linking is included, and the smart linker removes routines that aren't accessed. Compilation is controlled by the project file (similar to a make file) which contains information about the target executable. OS/2® developers should note that the default parameter passing convention is register-based. So, the first thing to do after installation is to modify the default project template to indicate the stack calling convention.

Brian Proffit

TopSpeed provides a multiple language development environment

The project file specifies the target operating system. There are compatible versions of the TopSpeed products for both OS/2 and DOS. So, developers can create DOS programs under OS/2 and vice versa. The project file also specifies the target result of the make process; any of the languages can be used to create programs, libraries, or dynamic link libraries. The project system is intelligent enough to detect if, say, a C++ program calls a Pascal function, and run the appropriate compiler. Memory model, runtime checks, and optimization level are also set in the project file. Note that these specifications may all be made from a pulldown menu. Knowledge of specifics regarding project file formatting isn't required.

In case your programs don't work correctly the first time — hard to believe, right? — the environment includes the Visual Interactive Debugger (VID). A full source-level symbolic debugger, VID works with all of the TopSpeed languages. Tracing through procedures called by programs written in a different language is no trouble. VID has very flexible breakpoint support, and includes the ability to watch variables and expressions for change during execution. Assembler code and register views are easy to access as well. The TopSpeed TechKit adds post-mortem facilities which integrate with VID.

To top off TopSpeed's features, the Environ-ment includes several useful utilities, includ-ing: a cross-file text search, — how many source files call the MungeWindow routine? — a file find utility for those whose directory tree is not as intuitive as they thought, an ASCII table, a calculator, a hex browser, and a keyboard macro facility.

So, the Environment is a nice place to work. But, what about the languages themselves? The first thing to note is that OS/2 API's, including Presentation Manager® API's, can be called from *any* of them. The bad news is that to do this you have to use the data types JPI defined in the runtime libraries which don't match the types listed in the OS/2 documentation. This makes the runtime library source essential to write OS/2 apps, since that's the only place the data types are documented.

C

This is a full ANSI implementation with extensions. Inline expansion of non-recursive functions is a nice feature. This allows the developer to use functions to make the source code more readable, without paying a performance penalty for the function call. Via pragmas, TopSpeed C allows specification of several types of runtime checking, including: dereferencing of null pointers, array index checking, and stack overflow. The library is re-entrant for multitasking applications. TopSpeed C is highly compatible with Turbo C ™ and Microsoft® C.™

C++

TopSpeed C++ is based on AT&T® C++ version 2.1,™ as described in *The Annotated C++ Reference Manual* by Ellis and Stroustrup. Extensions, accessible when not running in ANSI mode, include: nested comments, additional keyword modifiers (cdecl, pascal, near, far, and huge), and inline machine code.

The Rogue Wave Class Library, available separately, provides classes for things such as collections (standard types such as linked lists and queues, as well as Smalltalk™ types such as Set, Bag, OrderedCollection, and SortedCollection), B-Tree file access, string manipulation, and an extensive set of math classes.

MODULA-2

Being a fairly young language —Niklaus Wirth designed it in 1980 as a follow-on to Pascal— the only standard for Modula-2 is Wirth's book Programming in Modula-2. This implementation is faithful to that standard, with extensions. The extensions are substantial, including constructs to support object-oriented programming.

The Communications Toolkit provides modules that make developing programs for serial communications easy. This feature is available separately and includes support for the popular file transfer protocols. The B-tree Toolkit provides high-speed indexed database manipulation routines.

PASCAL

This is a full ISO level 1 implementation with a broad set of extensions when not running in ISO enforcement mode. As with Modula-2, there is a complete set of extensions to support object-oriented development. Other extensions include separate compilation (units), dynamic strings, and inline procedures.

A Turbo Pascal® to TopSpeed Pascal conversion program is included as well as units and pragmas to help support conversion of Turbo Pascal code.

SUMMARY

The acceptance of the TopSpeed product family can best be indicated by the number of vendors marketing products that work with it. Approximately twenty companies have joined with JPI to form the TopSpeed Consortium. The Consortium publishes a catalog listing over 50 products which work with the TopSpeed languages.

JPI has announced compatible 32-bit OS/2 versions of all products. So, whether you're developing for OS/2 32-bit, OS/2 16-bit, or DOS, TopSpeed products on OS/2 2.0 provide an excellent platform.

Happy developing!

Brian Proffit, *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, FL, 33429. Mr. Proffit is a senior programmer in OS/2 software developer strategy. He is currently responsible for OS/2 developer tools. He joined the development laboratory in Boca Raton in 1983 after working as a systems engineer in Dallas.*

FOR MORE INFORMATION:

Jensen & Partners International
117-F Hunt Hill Road, Unit 1
Rindge, NH 03461
800-448-4440; 603-899-6470

Jensen & Partners UK, Ltd.
Unit 21, 63 Clerkenwell Road
London EC1M 5NP
44-71-253-4333
FAX: 44-71-251-1442

Software Tools

FingerTips: A Real-Time OS/2 Application Development Environment

Tony Fazio

James W. Rascoe

by Tony Fazio and James W. Rascoe

FingerTips™ is an object-oriented OS/2® development environment for creating Presentation Manager®-based applications interactively in real time. Developed and marketed by Fortis Development Corporation of Chatham, New Jersey, FingerTips was previewed at the 1991 IBM® PS/2® Forums and was introduced formally to the OS/2 marketplace in October of 1991. It is designed for true 32-bit operation under OS/2 2.0.

PRODUCT OVERVIEW

FingerTips functions as an extension to the OS/2 operating system, enabling developers to exercise IBM's Standard and Extended Services directly — without coding or compiling. This interactive development capability is made possible by a centralized relational database repository that is used to store all definitions and logic for the PM applications under development. Side-by-side display of the developer workbench and the user application lets developers dynamically create and change program logic, and immediately see the effects in the working program.

In the PM environment, a working program is created by exercising logic against an application's window and dialog controls. So the PM application development task boils down to pointing the right logic to the right control at the right time and in the sequence required to perform all application functions. (The development process in the FingerTips environment is described later in this article, in the section "Building Application Logic with FingerTips".)

The approach taken by FingerTips is very straightforward (see Figure 1). The objects which comprise the FingerTips engine closely interface and communicate with the Standard and Extended Services supplied by IBM with OS/2. The FingerTips user interface provides a consistent structure for organizing all user classes and objects, and for using the FingerTips engine functions to bring an application's windows, menus, icons, dialogs and children immediately to life.

Ideally suited to the rapid development of On-Line Transaction Processing (OLTP) client-server applications, FingerTips offers OS/2 installations a number of significant advantages. FingerTips:

- requires no new language to be learned.
- requires no source code to be written or compiled.
- executes non-interpretively at compiled speeds both during development and at run time.
- eliminates the need for programmers to master PM APIs.
- provides complete program flow control.
- automatically handles all dynamic memory allocation and multithreading management.
- dynamically binds databases and executes SQL statements.

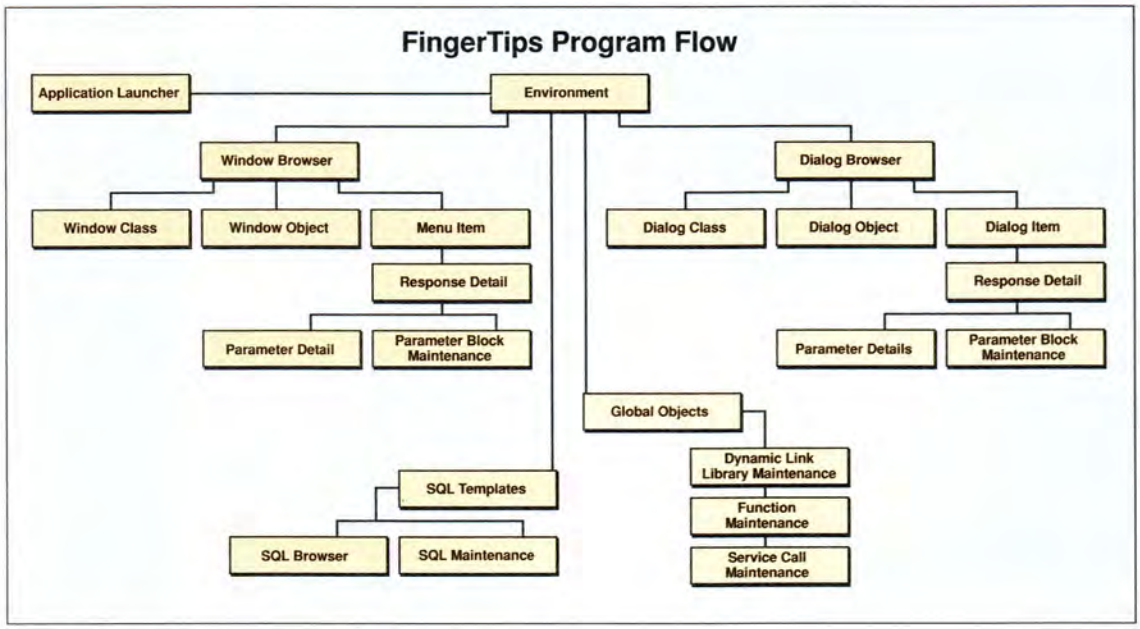


Figure 1. FingerTips Program Flow

In addition, the ability to add an unlimited number of user-defined functions to the FingerTips environment makes it possible for OS/2 installations to develop the full range of PM-based applications — from the simplest to the most complex — in real time, from start to finish.

THE FINGERTIPS ENGINE

The overriding objective in the development of FingerTips was to eliminate the need for programmers to tediously code, compile and re-compile repetitive API calls and other functions into their PM applications. Fortis developers solved this problem by writing these functions once (very compactly and efficiently), compiling them, and then storing them in one place — a relational database repository — where they can be called and used by multiple developers simultaneously.

In communicating with OS/2's Standard and Extended Services, the FingerTips engine requires detailed definitions and other application information that must be directly entered or imported into the repository. Once in the repository, this information is then available to the FingerTips user interface and can be used to build application logic interactively. The FingerTips engine manages the information that governs the four major components of PM applications: windows/menus, dialogs, database/filing systems, and communications.

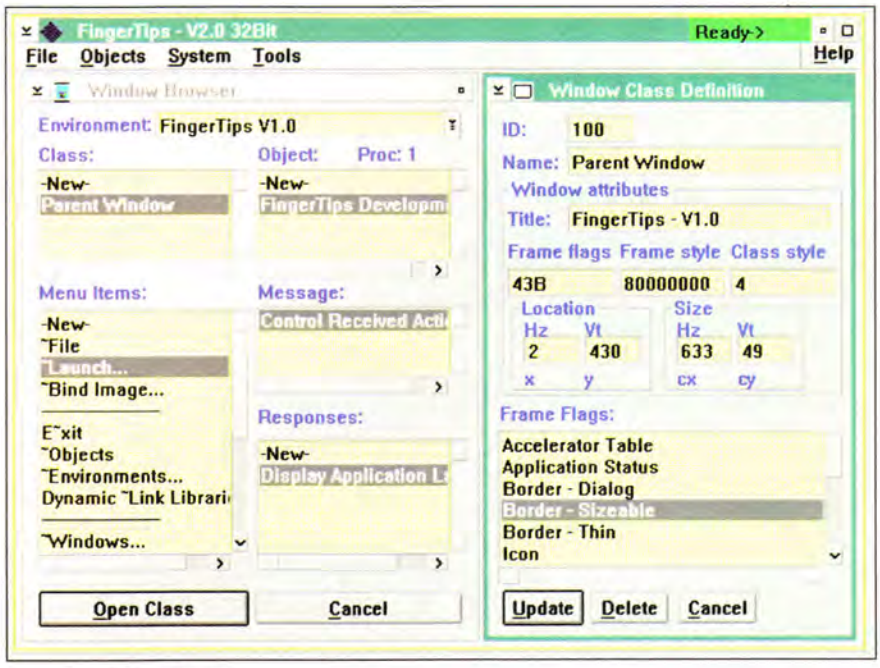


Figure 2. Window Creation

Windows and Menus

FingerTips lets developers interactively define all attributes of an application's windows and menus, as well as update and delete selected attributes in real time. Because FingerTips is fully SAA™-compliant and the repository always knows where all attributes are stored, FingerTips is unaffected by changes to Presentation Manager. FingerTips users can take advantage of the latest PM enhancements and revisions automatically, including most new Version 2.0 controls, with no need for an update to FingerTips. As PM grows in functionality and versatility, FingerTips will grow right along with it. (See Figure 2.)

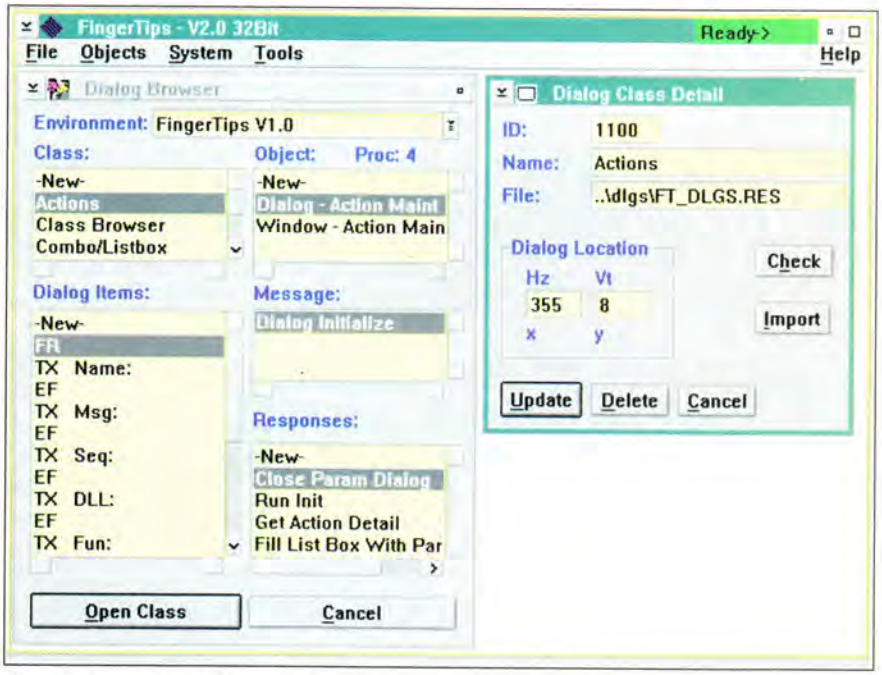


Figure 3. Import from Dialog Editor

Dialogs

FingerTips integrates seamlessly with the IBM Dialog Editor. This permits developers to import dialogs into the FingerTips environment quickly (see Figure 3). All attributes of each dialog control — its sequence, control ID, class, screen orientation and size — are stored in the FingerTips repository where they can be accessed and utilized readily via the user interface. FingerTips supports the inclusion of one or several dialogs in a given source file to meet any application requirement.

Database and Filing Systems

FingerTips supports concurrent communications with multiple databases and provides parameterized SQL capabilities. Developers can create their own SQL templates quickly, and they can dynamically change the SQL statements and parameters to be called in meeting new application requirements. FingerTips supports IBM Database Manager (Version 2.0) in the repository role; future releases will include support for a variety of other relational DBMSs and filing systems as well.

Communications

The FingerTips engine provides connectivity to a wide range of systems through its support of LU6.2 APPC and Remote Data Services. It houses information on the other types of systems with which PM applications will be communicating (e.g., AS/400® to SQL/400® or the native AS/400 filing system via PC Services, ES/9000™ to DB2® via CICS™, etc.). FingerTips' client-server architecture greatly facilitates the development of OLTP applications that must communicate and exchange information with systems running on diverse hardware platforms in different operating environments.

THE FINGERTIPS USER INTERFACE

The FingerTips user interface enables developers to access FingerTips functions and application information from the repository, and then use them in building their PM applications. An interactive trace facility is available for use in monitoring system operations. Plus, FingerTips users can access the IBM Dialog, Icon, and Font Editors from within FingerTips.

The Environment: Application Work Area

The first step in using FingerTips is to create an "environment" in which a developer's application information can be stored and organized. The user environment can house information on several applications and serves as the FingerTips developer's work area. FingerTips provides comprehensive facilities for creating and maintaining environments,

and for launching applications defined in them (see Figure 4). FingerTips also provides a versatile facility for importing and exporting windows, dialogs and other application information between environments. For example, source definitions can be exported from the repository into a variety of file formats (ASCII delimited, IXF, etc.) that then can be used as source code input to an installation's version control system. In addition, application information can be exchanged with FingerTips users in other departments or companies that do not utilize a centralized repository database.

Dynamic Link Libraries

The FingerTips engine includes a carefully designed set of core functions to facilitate the development of OLTP applications. In addition, FingerTips installations can create an unlimited number of user-defined functions, using any OS/2-supported language, to meet their unique processing needs (see Figure 5). All FingerTips and user-defined functions are stored as compiled code in Dynamic Link Libraries — re-usable objects that developers can invoke and to which they can pass parameters in creating their applications.

Creating Window/Dialog Classes and Objects

The object-oriented FingerTips user interface requires that a developer's windows and dialogs be grouped into classes, and that each window and dialog be defined as a specific object within its class. This is necessary in order to make these windows and dialogs accessible to FingerTips' window and dialog class "browser" facilities. These are the tools used to dynamically select application windows/dialogs and isolate the controls to be exercised by application logic. FingerTips provides easy-to-use facilities for defining window and dialog classes. Plus, it supports the creation of an unlimited number of windows and dialogs, and permits them to be shared among environments.

FINGERTIPS FUNCTIONS: STRUCTURE AND USE

To understand how FingerTips delivers its real-time development capabilities, one must first understand the very different way in which functions are structured and used in the FingerTips environment:

- Unlike traditional functions that perform a prescribed duty or service, FingerTips functions are designed to be capable of performing many different duties. The mechanism that makes this possible is the FingerTips "service call", which defines a specific duty that the function can be directed to perform.

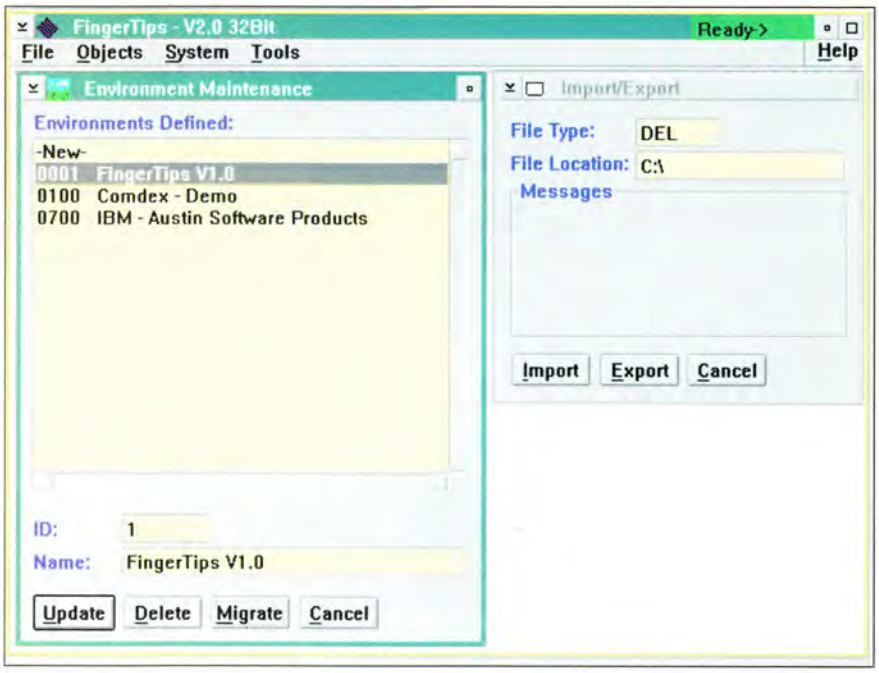


Figure 4. Environment Maintenance

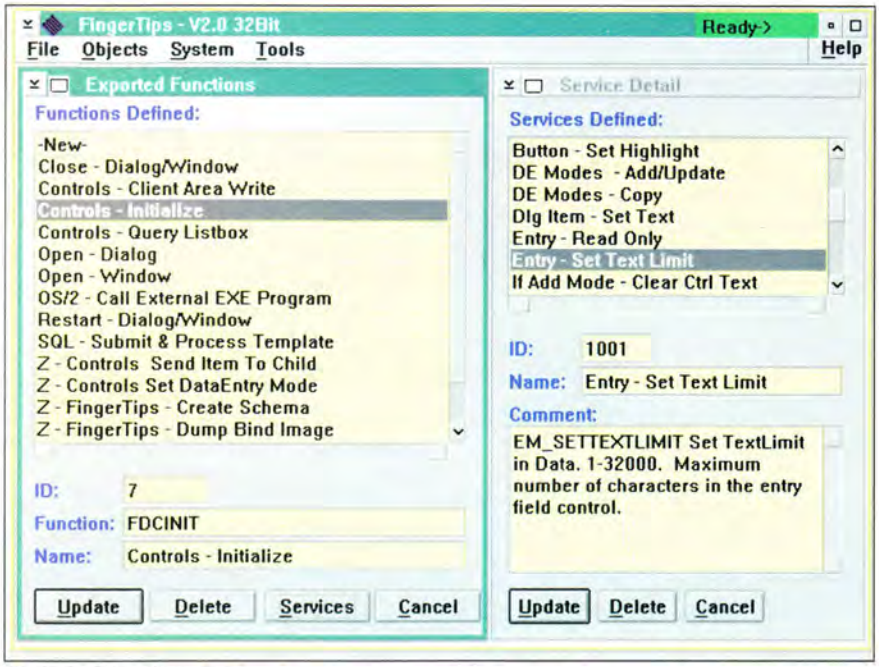


Figure 5. DLL Maintenance

- A function can contain an unlimited number of service calls. This gives users the flexibility to expand the capabilities of their user-defined functions to meet new application requirements, while enabling Fortis to add new services to its core system functions.
- A function will utilize only those services defined in parameters that are included in its parameter block. The FingerTips user interface enables the developer to select a specific service from those available within the function and direct it toward any control at any given point in time.
- An unlimited number of parameters can be passed to any function. If the same action (e.g., setting text length) must be performed repeatedly (e.g., for 50 different fields), it is no longer necessary to define and call that function repetitively each time it is to be used. The developer simply calls the appropriate function once and passes 50 parameters to it (which direct that action to the required 50 fields).

This capability dramatically reduces manual coding requirements and increases development productivity. Because it is only necessary to call a given function once, FingerTips also cuts CPU resource consumption and greatly enhances system efficiency.

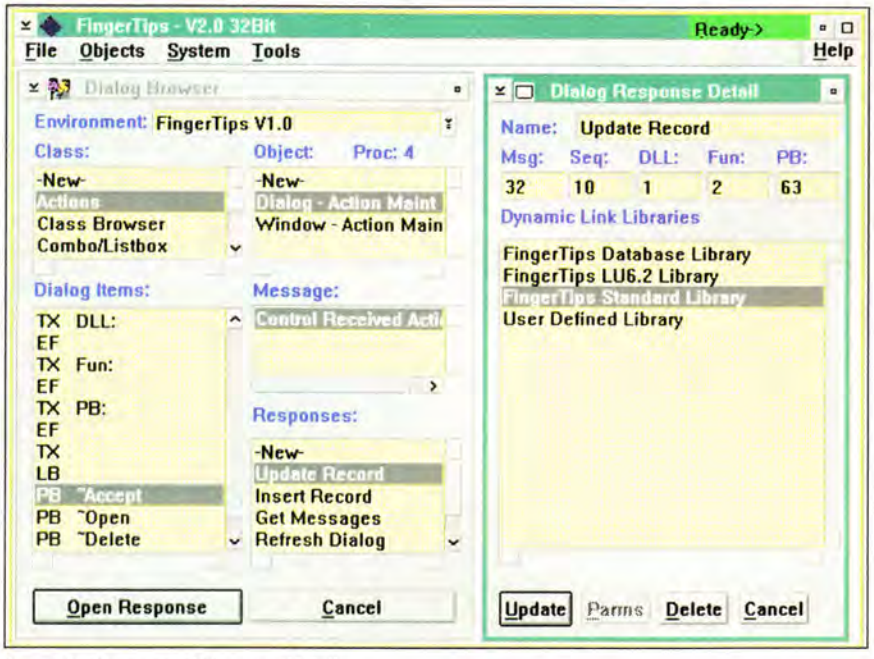


Figure 6. Browser and Response Detail

BUILDING APPLICATION LOGIC WITH FINGERTIPS

The process of attaching specific logic to specific controls is accomplished in FingerTips through the use of three interface panels: Window/Dialog Class Browser, Window/Dialog Response Detail, and Parameter Detail.

Window/Dialog Class Browser

- Use the window or dialog browser to isolate the control to be exercised by the logic (see Figure 6).
- Determine a response for the control and when it will be taken.
- Select that response by pressing Open Response button.

Window/Dialog Response Detail

- Use the window or dialog response detail panel to specify:

MESSAGE: The PM message which defines the condition or point in time when the response will be taken.

SEQUENCE: If multiple responses are to be taken, this specifies the order of this response in the sequence.

DLL: The FingerTips or user-defined DLL that contains the logic or function to be exercised.

FUNCTION: The specific function within the DLL that performs the desired response.

PARAMETER BLOCK: The parameter or group of parameters to be passed to the function that will enable it to make that response.

Parameter Detail

- Use the parameter detail panel to pass all parameters to the function (see Figure 7). FingerTips requires a fixed format for every parameter that is passed to every FingerTips function. Each must contain the following elements:

SEQUENCE: If multiple parameters are to be passed, this specifies the order of this parameter in the sequence.

SERVICE CALL: The specific service available within the function that will be used in passing the parameter.

ENVIRONMENT: The specific environment in which the object to be affected by the parameter resides.

TYPE: The type of object (window, dialog or SQL template) to be affected by the parameter.

OBJECT: The specific window or dialog to be affected by the parameter.

ITEM: The specific child (if any) on the window or dialog to be affected by the parameter.

DATA: Any user-defined information (e.g., messages, flags, etc.) that must be passed to a control.

- Exercise the target control. The desired response has now been attached to the control and it will be made when the control is exercised.

- Proceed to attach the required logic to the remaining controls on the application's windows and dialogs.

It is important to note that all application logic attached in this way is stored directly in the repository. This means that the developer not only is able to create the application in real time, but now can change and maintain it in real time, as well, with equal ease. There is no need to restructure internal program logic or rewrite source code. All that the developer needs to do is isolate the controls that must be changed and attach different logic and/or different parameters to them. The changed logic is stored in the repository instantly, and the modified application begins executing immediately.

Finally, developers should note the robust development potential that has been built into FingerTips. The five response detail options and the seven parameter detail options described above may be combined in literally thousands of ways to meet virtually any application requirement. This same robustness is available to FingerTips developers in building their own user-defined functions.

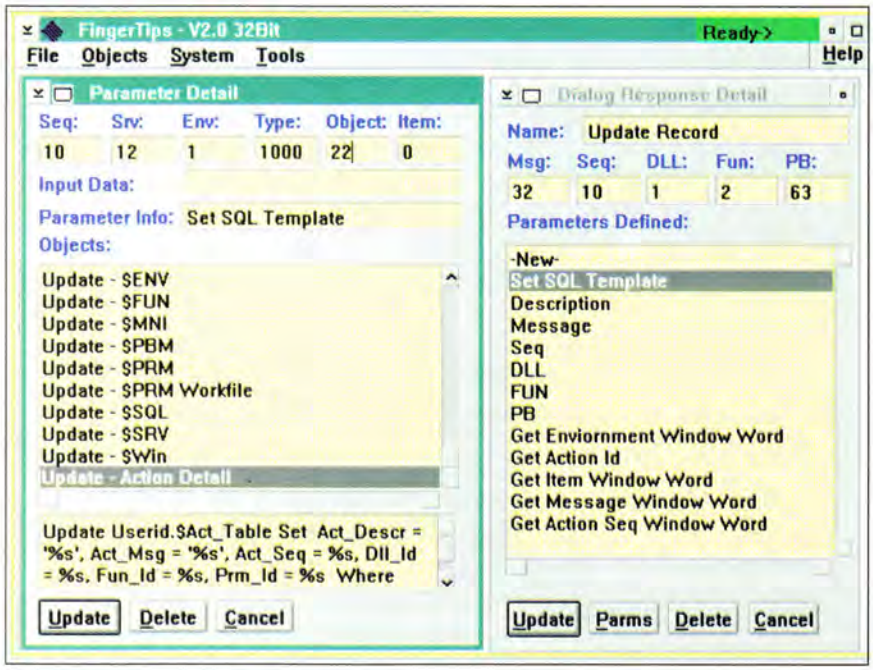


Figure 7. Response with Parameter Detail

DISTRIBUTION AND RUNNING OF COMPLETED APPLICATIONS

Once an application has been debugged completely and is ready to be moved into production, FingerTips generates bind images of the application's parameters and other attributes and combines them into a resource file.

James W. Rascoe can be contacted on USSWP001 IBMMAIL

In the FingerTips environment, the resource file serves the same function as code in a traditional program. It not only passes needed parameters to the FingerTips functions, but it also supplies all screen attributes and operational characteristics. It defines where each field is to be associated on a given dialog or panel and provides the logic that describes what operations each field is supposed to perform. The resource file then attaches to each control a PROC address to the appropriate real function (which can reside either in the FingerTips DLLs, or in the OS/2 DLLs themselves) that must be executed.

For distribution purposes, the resource file can be stored as a permanent entity on either hard disk or diskette, and it may contain the bind images of one or several applications. The resource file is combined with the FingerTips EXE file and DLLs (both FingerTips and user-defined) to form the production version of the application (which no longer requires the FingerTips development tools or repository). At run-time, the EXE file brings the DLLs and resource file into memory where they work "hand-in-glove" with the OS/2 DLLs and begin executing immediately at compiled speeds.

THE FINGERTIPS DIFFERENCE: WORKING APPLICATIONS WITHOUT CODING

FingerTips is the first OS/2 development system that can deliver a fully functional, working PM application — from initial window and dialog definition to fully tested and debugged production program — in real time, without use of a programming language or other application development tools. What's more, it runs non-interpretively and at compiled speeds throughout development. And developers are freed from the need to master PM APIs, manage multithreading, and perform dynamic memory allocation.

An inherent "proof" of the power of FingerTips can be seen in the fact that the FingerTips user interface is itself a FingerTips application. It passes parameters — selected and combined to create the interface environment — to the same FingerTips functions utilized by other applications. It was built in exactly the same way that any other FingerTips application would be built, without coding or compiling.

FingerTips does make some demands on the developer that other products do not. Developers must come to FingerTips knowing precisely what it is they want to do, what processing must be done, and how the program must flow from an end-user's point of view. They must understand the data that they either have available to them or will be creating in the course of the application. And they must have a firm understanding of PM controls — what they can do, how they should be used, and how to combine them on dialogs in a way that will navigate the end user quickly and efficiently through the application.

Developers armed with this understanding will find FingerTips a uniquely powerful OS/2 application development environment. With FingerTips, developers can leverage all the power of OS/2's Standard and Extended Services, in real time. The result is superior OS/2 application development speed and flexibility.

Tony Fazio, *Fortis Development Corporation, P.O. Box 299, Chatham, NJ 07928-1719, (201) 635-5700. Mr. Fazio is co-founder and President of Fortis Development Corporation. He has 12 years of experience in systems integration and technical sales, and today directs all Fortis marketing and administrative activities.*

James W. Rascoe, *co-founder and Vice President of Fortis Development Corporation, is the chief engineer and developer. He has 15 years of experience in designing software products and developing applications for multiple platforms. Mr. Rascoe can be contacted on USSWP001 IBMMAIL.*

Software Tools

GammaTech Utilities for OS/2 2.0

by Benny N. Ormson

Information presented in this article is based on anticipated features of the GammaTech Utilities version 1.30. This version was in development at the time this article was prepared and may differ from the actual delivered product which is expected to be available near the end of 1991.

The OS/2® High Performance File System (HPFS) offers many advantages over the old DOS FAT file system. But most utility software developed for the FAT file system will not work with HPFS. This incompatibility is due to significant differences in file system structures.

The GammaTech Utilities for OS/2 allow you to: undelete files, optimize volume performance by defragmenting files, search a volume for files, list directories, view and set file attributes, edit sectors on your drives, test media for errors, obliterate sensitive data, display fragmentation and error information, and remove undesired files in mass. In this article I will describe the various components of the GammaTech Utilities and explain the importance of each to file management.

WHY OPTIMIZATION IS REQUIRED EVEN WITH HPFS

While HPFS's file management and caching techniques tend to minimize fragmentation and its effects, there are still cases where severe fragmentation can occur. The worst examples are aged volumes wherein free space fragmentation is a natural result of files being randomly created and erased over a period of time. As a result, HPFS has a large pool of free space to choose from when a new file is created.

Benny N. Ormson

Another fragmentation condition occurs when data is appended to a file. While the original file may be contiguous, data added at the end is often placed in another free area on the disk, thus fragmenting the file.

The OS/2 file APIs provide the developer with the option to tell HPFS how large the file will be. HPFS will then choose a free space area consistent with the predicted file size. If the developer underestimates the file size, fragmentation is the likely outcome. When a developer does not provide this information to the API, the HPFS must make a guess. The outcome is likely to be an underestimate. If the developer over-allocates the estimated file size, data appended later will remain in the same contiguous file space. The drawback to this option is that free space may be wasted.

Since these OS/2 features were not available under DOS, DOS programs executed under OS/2 often generate fragmented files. Poorly done ports of DOS programs to OS/2 often experience the same problem.

The Optimizer

The HPFS Optimizer utility examines each file on the HPFS volume to determine the number of fragments the file contains. Highly fragmented files are then re-saved to disk as contiguous (unfragmented) files or as files with fewer fragments. The Optimizer allows the user to specify the minimum and

maximum number of fragments a file must have to be considered for optimization. If no specification is made the Optimizer's default is a minimum of three. Once the Optimizer begins, the multitasking and file sharing capabilities of OS/2 allow it to run while other processes are active on the machine. Open files, in use by other processes, are bypassed.

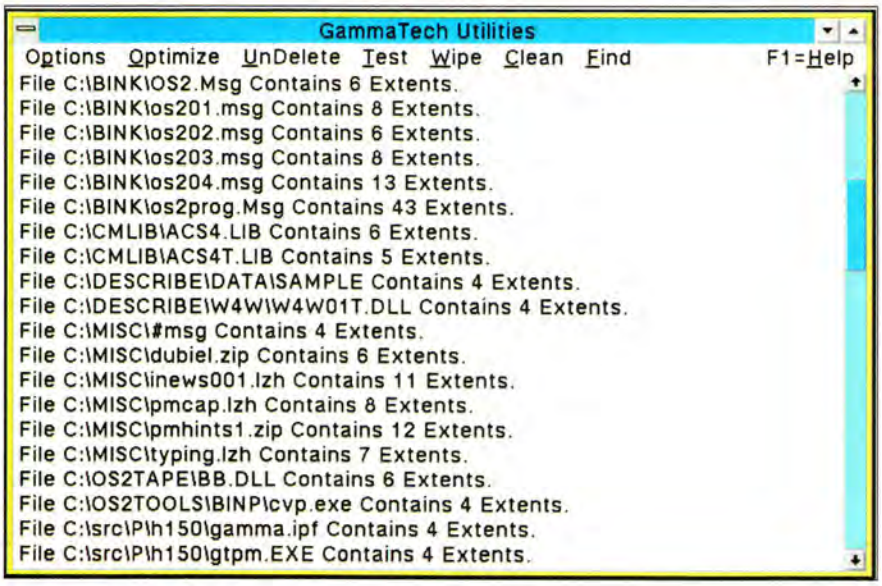


Figure 1. The HPFS Optimizer

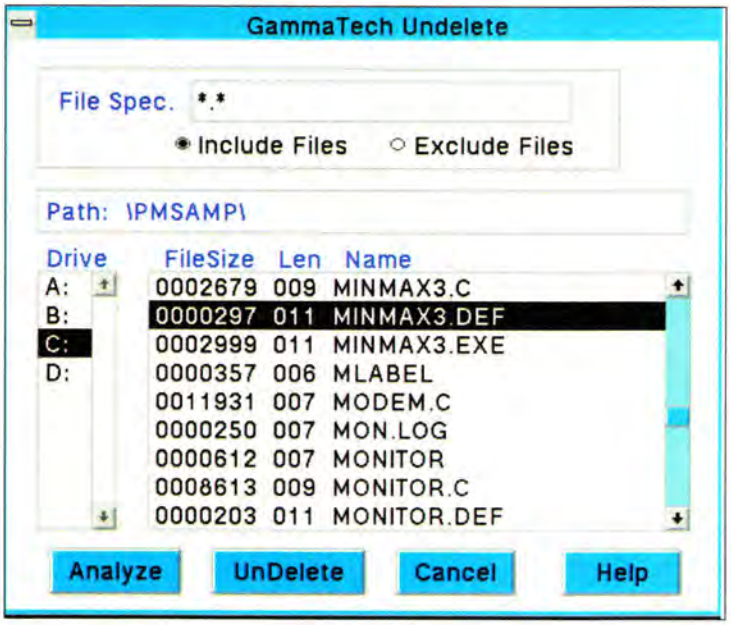


Figure 2. The UnDelete Utility

The defragmenting of files naturally increases the fragmentation of free space on the HPFS volume. This is a result of replacing a large contiguous free space area with unfragmented files and returning several smaller areas to the free space pool. The default value of three fragments seems to give a good balance in that it provides relatively minor file fragmentation without generating excessive free space fragmentation. HPFS's excellent caching techniques minimize the effect of small degrees of file fragmentation.

The UnDelete Utility

The UnDelete utility is a Presentation Manager™ application which will analyze an HPFS or FAT volume to locate all files which may be at least partially recovered. The files are listed and may be individually selected for recovery.

When an erased file is recovered, the UnDelete utility will prompt for the new output file name and will attempt to direct the recovered file to a different volume than the volume where the erased file resides. This prevents the newly recovered file from overwriting the data of the erased file before it has been recovered. The user can force UnDelete to recover to the same volume but the chances for a successful recovery of all data is reduced.

The HPFS file system structure lends itself to much greater reliability in file recovery than the FAT file system. The HPFS method of banding data areas and using different bands for concurrent output files also improves the chances of a later file recovery, even after new files have been generated on a volume after the file was erased. The HPFS method of space allocation insures better data tracking and makes file recovery more reliable.

File Recovery

OS/2's HPFS, with its long file names, extended attributes and the multitasking environment of OS/2, changes the file recovery process. The DOS FAT-based recovery utilities enjoy the ability to gain exclusive access to the disk drive, allowing direct modification of it. In a multitasking environment, multiple processes can access the drive simultaneously. Direct updates to the

volume in this environment could cause loss of integrity. For this reason, the GammaTech file recovery utility does not reconstruct lost data in place on the volume. The recovery utility creates a new file from the erased file data. Since the new file data could overwrite the erased file data during the recovery operation, the utility defaults to writing the recovered file to a different volume than the volume where the erased file resides.

The OS/2 file system allows attaching information to files using extended attributes. Examples of such information are file comments and icons. The extended attribute data is often stored separately from the file data. The file recovery procedure attempts to account for any extended attributes belonging to the erased file. However, there are times when an erased file's data can be recovered though the extended attributes for the file have been destroyed.

Another feature of HPFS is the long file name convention which allows file names to be up to 254 bytes in length. Although the HPFS structure provides greater integrity for recovering files, the file names are not entirely intact if they exceed 15 bytes. In this case, the user must identify the correct file to recover based on the first 15 characters of the file name, the total length of the file name, and the size of the file.

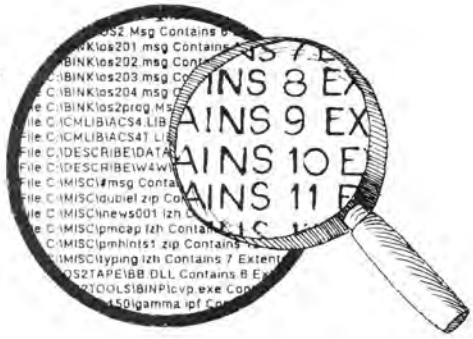
HPFS maintains its directory structure in a manner which maximizes performance. Directory entries for erased files are usually removed immediately. While simply scanning the directory B-Trees for erased files is fast, many erased files can't be located. The GammaTech file recovery functions actually scan the free space on the volume for lost files. Most of the required information needed to recover a file is located in the F-Node block which precedes the file data. By searching free space for erased F-Nodes, virtually all erased files which can be recovered are located. The drawback to this technique is that the analysis of the volume can require several minutes, particularly if the volume contains a large amount of free space.

TEXT MODE VS. PRESENTATION MANAGER

Some of the utilities contain Presentation Manager® and text mode versions. Several others are text mode only. Beside the fact that some operations are just plain easier, faster, and simpler to perform from the OS/2 command line, text mode allows the utilities to be used in maintenance mode.

Maintenance mode would be used when the system cannot or should not be booted from the hard disk due to an error. The user would then boot in maintenance mode, correct the problem, and then reboot normally. Since Presentation Manager is not available in maintenance mode, several of the utilities are supplied in text mode form.

Maintenance mode is achieved by pressing Esc from the logo screen when booting from the OS/2 installation diskette. This provides the user a command line prompt. The installation diskette is removed from the drive and a copy of the GammaTech Utilities diskette is inserted. The text mode versions of the Utilities are then available for use.



Locating Files

Files can be located using the text mode Where utility or the Presentation Manager Find function. The user has extended capabilities over and above the standard utilities provided with OS/2. Searches may be performed on multiple HPFS and FAT volumes based on file names, comment text patterns, and file time stamps. Searches may be started at the root directory or any specific path. All searches include subdirectories of the specified path and will locate hidden and system files.

The List Directory Utility

The List Directory (LD) utility is a command line utility that displays directory information. It provides over 25 user selected options which allow users to tailor the output to individual tastes. These options include: sort order, case of directory and file names, content of the output, and many more.

The LD utility has several unique features. The user can display file fragmentation, file attributes, extended attribute sizes, all three date/time values for HPFS files, and volume information. Hidden files may be displayed. The /S option will include output for all subdirectories.

Figure 3. The Sector Editor

Comments may be added, displayed, and deleted from files and directories. These comments are compatible with the file comments provided with the OS/2 File Manager. Comments added with the LD command are visible to the File Manager and comments added with the File Manager™ are visible to LD. These comments will remain with the file even when it is renamed or moved and are automatically removed when the file is erased.

The user can set default values for the various options using an environment variable or simply use the standard default values. Any option can be overridden on the command line when the command is executed.

The Attribute Utility

The Attribute utility has several features. It can be used to alter any of the standard file attributes of a file. It provides greater flexibility than the OS/2 File Manager in that it will alter the system attribute. The user can modify any of the three HPFS file date/time values. Options provide the user with the ability to set these date/time values to specific values, to the current date and time, or set one timestamp equal to another timestamp. Multiple files and subdirectories may be modified in one execution by using wild card values in the file name.

The Sector Editor

The sector editor provides low level access to raw sectors on hard disks and floppy disks. Operating independent of the media format, the sector editor provides ability to alter otherwise restricted areas dealing with the file system format. Thus, its use should be limited to those persons with technical expertise in file system formats.

Access is provided to physical drives, logical volumes, and individual files. The sector editor can also be used to examine and modify the master boot sector, partition table, etc.

Miscellaneous Utility Functions

Several additional features are provided with a Presentation Manager implementation which allow users to more easily maintain their workstation disk volumes. Media testing of specified volumes, HPFS or FAT, may be performed. All sectors of the requested volume(s) are tested for errors. Information regarding specific FAT or HPFS volumes may be displayed as well as unique information, such as the status of HotFix area for HPFS volumes.

The Wipe feature, which utilizes the free space option, can be used to obliterate previously erased files so they cannot be recovered.

A Clean feature is provided which allows the user to mass erase selected files such as all files with a ".BAK" extension. An option is provided to erase all files which have a zero data length.

SUMMARY

The GammaTech Utilities for OS/2 provide capabilities for the OS/2 workstation environment which allow maximum performance, data recovery and user productivity. Its Presentation Manager, SAA™ -complient, implementation provides an easy to use platform for maintaining the workstation.

REFERENCES

Ruley, John D. **Windows and OS/2 Magazine**, *Good News for OS/2 Fans*, June 1991, Volume 2 Number 4, page 42.

Demo of the 1.3 version:

IBM Field Television Network Broadcast, *OS/2 File Systems*, November 15, 1991. Available from IBM SE on VHS video tape.

Benny N. Ormson, *GammaTech, P.O. Box 70, Edmond, OK 73083, (405) 359-1219. Mr. Ormson has been active in data processing for 18 years. He has primarily worked writing large system utility and telecommunications systems programs for Southwestern Bell Telephone Company. Mr. Ormson founded GammaTech in early 1991.*

Database Manager

Tackling Dynamic Panels and Queries in Query Manager

Theodore Shrader

by Theodore Shrader

IBM's OS/2® Extended Services is composed of new releases of Database Manager and Communications Manager. This latest release of Database Manager (and its supporting applications: Database Tools, Query Manager, and the DBM Command Line Interface) is designed to run on OS/2 SE version 1.30.1 or above, and OS/2 2.0. It also will run on base operating systems on OEM hardware that are equivalent to OS/2 SE version 1.30.1 or above, or OS/2 2.0. Database Manager has been enhanced to include such features as roll forward recovery (to recover lost on-line data), a Database Command Line Interface, SQL date and time arithmetic, Windows™ database client support, and user-defined collating sequences. The Database Command Line Interface allows users to run SQL commands as well as other database commands either from the OS/2 command prompt or from a batch file. Database Tools contains the Directory Tool, Configuration Tool, and Recovery Tool, each its own PM application. The Directory Tool allows users to manage workstation access to databases through Database Manager directories, by adding and removing databases and workstations from these directories. The Configuration Tool lets users customize resource parameters for a database or the Database Manager. Lastly, the Recovery Tool provides a user interface to the backup, restore, and roll forward recovery of a database.

The latest version of Query Manager does not use any internal interfaces to the Database Manager. It has matured as a query product that not only handles ad hoc queries, but also allows users to create applications with panels, menus, queries, and report forms. This ability to create and customize applications makes it unnecessary for end users to learn the structure or syntax of the SQL language.

Query Manager continues to take advantage of the SQL DDL and DML commands available in the latest version of Database Manager. This is the case especially in the realm of dynamic panels and queries. The following application program example demonstrates how Query Manager can adapt to suit the nature of existing data by presenting panels dynamically. The application program also helps to ensure consistency beyond the referential and field checks made by Database Manager, and allows users to view information within the context of their data by dynamically constructing SELECT statements.

BUILDING THE TASK MANAGEMENT APPLICATION

For example, let's say a development group wants to use Query Manager in the client-server environment to collect project management information from all of their developers. This information would center around tasks: who will be performing the task, what phase the task will be in, its description, and the dates this task will span. With developers using Query Manager at their workstations, all the data would be stored in the Database Manager at the server. At a later time, the development leader could use a specialized project management tool to extract the data from the database and create PERT charts and other information for resource management. The problem for the development leader would be how to create the interface for all of the developers on the project and make sure the data remained consistent for

the application. For one, these consistency checks would make sure the start date for a task was not after the task's end date.

To enable dates to be tracked adequately, each task must have a start date and end date pair associated with the planned, revised, and actual task dates. The planned date pair would be the dates that the developer initially forecasted the task to span over. The developer would not have to enter both planned dates at one time, but once both dates were entered, they would be copied to the revised date fields. Thereafter, the developer would not be able to change the planned dates. This would allow the team leader to track date movement. Developers could alter revised and actual dates anytime after the planned dates were recorded to reflect their current situation. Revised dates give developers the opportunity to change their initial date ranges, if needed, and actual dates naturally reflect when the task was started and completed.

By using these rules, the application can construct a consistent framework from which to enter dates and to run queries. For example, always having a pair of revised dates means that queries examining the dates of uncompleted tasks can focus on the revised dates instead of the planned or actual ones (which may not be complete or up-to-date). Allowing developers to enter their planned start and end dates at different times gives them the opportunity to enter the task in the database (to make sure it is accounted for), but it also allows them to commit to a set of dates at a later time. Lastly, uncompleted tasks can be separated from completed ones by checking to see if the actual end date is null or not.

Panel objects in Query Manager can be executed in three modes: add, change, and search. Panels in add mode take new information and insert their collection of fields as a row into a database table or view. Change mode allows users to modify existing rows and search mode gives users the ability to retrieve a row or set of rows based on specified criteria.

This application example can implement the date rules by creating three different panels for the user. The first panel covers the add mode, and contains all the possible fields the user could enter, except for the revised and actual date fields. As per our application rules, these other fields are available to the user, but only in change mode, and only after the planned dates have been entered and the data row stored in the database.

The other two panels run only in change mode. For the case where the planned dates have been entered, the second panel displays those planned dates, along with input fields for the revised and actual dates. The third panel handles the case where a task record was entered, but without the planned start and end date pair. It is similar to the add panel in that it prevents the user from entering the revised and actual dates by not showing these fields.

To retrieve data, the application will need, as a minimum, a generic query that can take varying user input, construct an SQL statement, execute it against the database, and return the information to the user. The example application handles this case, and additional queries can be constructed using these panels and procedures as templates.

CREATING THE DATABASE TABLES

Before constructing the panels or queries, the application first must create the Database Manager objects that will be used as the base objects to store data entered by the developers or users. The application will use three tables: TASK, NEXT_ID, and TASK_QUERY. Figure 1 shows the CREATE TABLE statement for the TASK table.

```
CREATE TABLE TASK
 ( TASK_ID SMALLINT NOT NULL, TASK_NAME VARCHAR(25) NOT NULL,
   PHASE VARCHAR(15) NOT NULL, DESCRIPTION VARCHAR(50), PSTART DATE,
   PEND DATE, RSTART DATE, REND DATE, ASTART DATE, AEND DATE,
   USER_TOUCHED CHAR(8), DATE_TOUCHED DATE, PRIMARY KEY ( TASK_ID ) )
```

Figure 1. CREATE_TASK Query

The TASK table is the main table of the application. It stores the task id (used as the primary key), the name of the task, phase, description, and the previously mentioned date pairs. By including the phase as part of the task record, the application can subdivide a task into its different parts, such as design, implementation, and testing. In addition, two administrative-type fields have been included: the date the record was added or changed, as well as the name of the user who modified it. These fields will not be shown on the panels, but they will be part of the reports of queries, giving the team lead and others an idea of who added or changed a task record, and when.

The NEXT_ID table is another administration tool. It reduces some of the overhead when adding tasks by working with underlying panels and queries to automatically assign and increment the TASK_ID field. This saves the user from having to enter a task ID, and thus also prevents the user from trying to duplicate an existing TASK_ID. (With a primary key on the TASK_ID column, Database Manager insures that a record is not stored with a duplicate ID. Further, by shielding the user from the column when adding the task, the potential for submitting an incorrect row is reduced.)

The TASK_QUERY table essentially is a copy of the TASK table with the additional USERCHG field and the START_TOUCHED and END_TOUCHED fields replacing the DATE_TOUCHED fields. The use of the NEXT_ID and TASK_QUERY tables will be explained later.

CONSTRUCTING THE QUERY MANAGER OBJECTS

Having created the Database Manager objects, the application now can implement Query Manager objects to use them. The first two objects will be the main procedure and menu. The main procedure will be the procedure the user runs from Query Manager or the OS/2 command line to start the application. It sets up the initial variables and runs the main menu. This menu presents the user with the options of adding a new task, changing an existing one, or querying the task table.

Adding Tasks

The add task panel is similar to one that can be created with the default definition action when constructing the panel, but with some important distinctions. As mentioned before, the task ID, date touched, user name, revised date field, and actual date field are not shown. Also, the add actions (add and next or add and keep) call different procedures instead of the Query Manager panel commands of "add" and "add and keep". These new add procedures take care of administrative tasks, and also check to see if both planned dates have been entered as well as if they are valid.

When the user selects add and next or add and keep, the TASK_ADD panel calls the TASK_ADD_NEXT or TASK_ADD_KEEP procedure. The procedures differ in what occurs after the row is added; add and keep retains previously entered information to be used as a template for the next add. However, both procedures call the TASK_ADD procedure, check to see if the return code was successful, and if so, add the row. They also increment the NEXT_TASK# field in the NEXT_ID table if the add operation was successful.

The TASK_ADD procedure, shown in Figure 2, runs the GET_TASK# panel. The user never sees this panel because it is run under the covers with an initial query to retrieve the current task ID from the NEXT_ID table, and an ending procedure that increments the value and places it into the task record.

When control is returned to the TASK_ADD procedure, it makes sure the TASK_ID field has a number, and if not, the field for the TASK table as well as the NEXT_TASK# field in the NEXT_ID table are initialized to one. (When called, the INIT_TASK query initializes the NEXT_ID table.)

The TASK_ADD procedure then checks to see if both planned dates have been entered. If so, they are copied to their corresponding revised dates. It also utilizes the date and user touched fields for administration purposes.

```
/* TASK_ADD procedure - main add procedure */
/* Get next task number and store it in the new_taskid global var. */
'run panel get_task# (mode=change)'
/* If the next_id table has not been initialized,
   initialize the next_task# field and the panel taskid field to 1.
*/
'get global (q_new_taskid = new_taskid)'
if (q_new_taskid = 0) then
   do
      'run query init_task#'
      'set current (taskid = 1)'
   end
/* Clear out any old revised dates. */
'set current (rstart = "")'
'set current (rend = "")'
/* Check on planned date pair. */
'get current (q_pstart = pstart)'
'get current (q_pend = pend)'
/* Copy over the planned dates, if both exist. */
if (q_pstart <> "") & (q_pend <> "") then
  do
   if (q_pstart > q_pend) then
    do
     say "The Planned Start date must be before the Planned End
date."
     exit 1
    end
    'set current (rstart = q_pstart)'
    'set current (rend = q_pend)'
   end
/* Initialize user and date touched fields. */
'get current (utouched = sqluser)'
'set current (utouched = utouched)'
'get current (dtouched = date)'
'set current (dtouched = dtouched)'
```

Figure 2. TASK_ADD Procedure

Dynamically Changing Tasks

This part of the application demonstrates how Query Manager handles dynamic panels. The TASK_CHGDAT panel (shown in Figure 3) handles rows where planned dates have been entered. The planned dates in this panel are displayed in read only format. The TASK_CHANGE panel takes care of cases where the planned date pair has not been entered by providing fields for the planned dates but not for the other date pairs. The application will display the TASK_CHGDAT panel in search mode since it offers all of the fields from which to narrow the search down. By not specifying any criteria, all of the rows are retrieved one row at a time.

Once the user selects the search action, the application must determine which panel to display. The instance procedure, CHECK_DATES, accomplishes this by setting up two variables. The first, which_pnl, stores whether the TASK_CHGDAT or TASK_CHANGE panel has been displayed. The second variable, gl_search, determines which action should be performed next, such as next, search, or exit the panel. The planned dates are returned and the task id is stored into the global variable gl_taskid. If either of the planned dates is empty, the TASK_CHANGE panel must be displayed by laying on top of the TASK_CHGDAT panel, which is already in the window. The procedure continues by flipping the which_pnl value and calling the

TASK_CHANGE panel. Upon its return, the gl_search variable is tested to see which action should be performed.

The TASK_CHANGE panel calls the TASK_CHANGE initial query which populates the panel using the gl_taskid variable that was stored in the CHECK_DATES procedure. Whatever the user's action, the application eventually will quit this panel and return to the underlying TASK_CHGDAT panel. The cycle continues for subsequent rows.

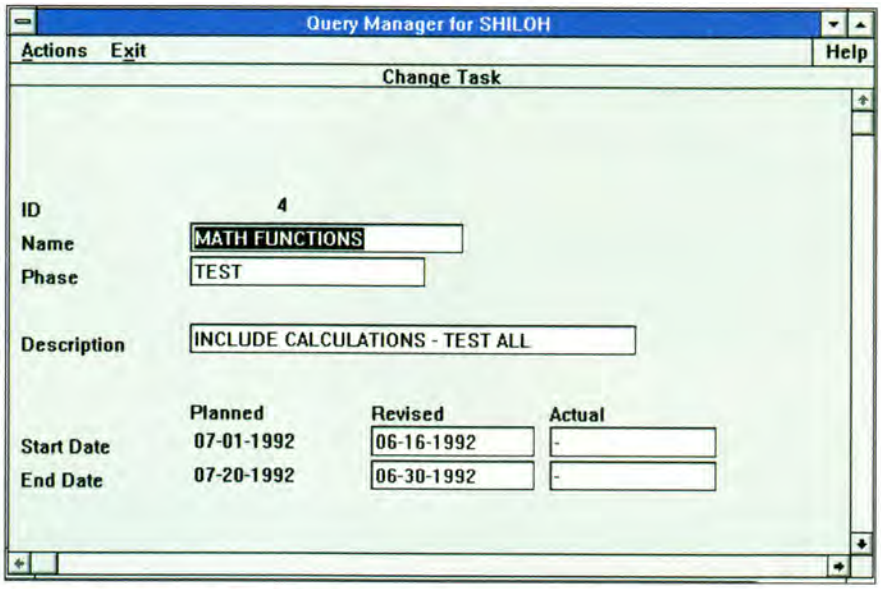


Figure 3. TASK_CHGDAT Panel in Change Mode

Whatever the panel, the change and next and delete and next actions will call the TASK_CHANGE and TASK_DELETE procedures instead of the default panel commands. The TASK_CHANGE procedure sets the gl_search value, changes the date and user touched fields to their correct values, retrieves the planned and revised dates to see if one or both of the planned dates need to be copied to the revised date fields, and checks to see if the entered date pairs logically follow each other. (Consult the TASK_ADD procedure to see how these checks are made.) Next, the change action is performed and the TASK_CHANGE panel is dropped if it is present. The TASK_DELETE procedure performs similar actions except the touched fields do not need to be updated or the date pairs checked. And of course, the row is erased instead of changed.

Although the TASK_CHGDAT panel can use the standard next and search panel commands, the TASK_CHANGE panel must have different ones to account for its "overlaid" state. Each procedure specializes in setting the gl_search variable to its current action before quitting the panel.

Dynamically Querying Tasks

When the user wants to query the tasks, the TASK_QUERY panel is called in add mode using TASK_QUERY as its root table. By using a separate table, Query Manager can be tricked into saving the user specified query fields, and then deleting them to prevent the "Panel has not been saved" pop-up from appearing. The start touched and end touched fields are also new to give the user a way of specifying an upper and/or lower bound on the date the record was entered or modified. For example, the user could ask for all records added or changed between 10-11-1991 and 03-05-1992 through this query.

By calling the panel in add mode instead of change, the application immediately can get the user specified criteria, dynamically build and execute the SQL SELECT statement, and create a report of the results. If the user went through the regular search and change modes of a panel, only one new row would be displayed at a time.

When the user enters the search criteria and selects the View Report action, the panel calls the TASK_QUERY procedure, shown in Figure 4. This procedure first gets the contents of all the panel variables and puts them into local variables. As described earlier, the panel values are saved temporarily in the TASK_QUERY table and then deleted using the GENERIC_DELETE_ROW query. Next, the global variables of inqry1 to inqry20 are set according to user specified search criteria. These variables are used in the GENERIC query which is a SELECT statement built with the inqry global variables. To build the query, each local variable is checked and if a value is present, the procedure creates a subclause to the SELECT statement and stores it into a unique variable. As a result, if the user did not

enter search values, a default report would be generated using the contents of inqry1 to inqry5. After all the local variables have been checked, the procedure sets the final local variable to get the tasks ordered, and finally, the procedure calls the GENERIC query.

```
/* TASK_QUERY procedure - for the TASK_QUERY panel */
/* get variables from fields */
'get current (q_taskid = taskid)'
'get current (q_taskname = taskname)'
...
'get current (q_astart = astart)'
'get current (q_aend = aend)'
/* create dummy row to delete in the dummy_report table
   this prevents the save data pop-up from appearing */
'BEGIN WORK'
'set current (taskid = 0)'
'add (type=cont, message=no)'
'run query generic_delete_row'
'set current (taskid = q_taskid)'
'END WORK'
/* clear and set global fields */
'set global (inqry1 = "SELECT TASK_ID, TASK_NAME, PHASE, DESCRIPTION,
")'
'set global (inqry2 = "PSTART, PEND, DAYS(PEND) - DAYS(PSTART), ")'
'set global (inqry3 = "RSTART, REND. DAYS(REND) - DAYS(RSTART), ")'
'set global (inqry4 = "ASTART, AEND, DAYS(AEND) - DAYS(ASTART), ")'
'set global (inqry5 = "USER_TOUCHED, DATE_TOUCHED FROM TASK ")'
'set global (inqry6 = " ")'
...
'set global (inqry20 = " ")'
addwhere = "WHERE"
addand = " "
/* construct query */
if (q_taskid <> "") then
   do
      inqry6 = addwhere || addand || "TASK_ID = " || q_taskid
      'set global (inqry6 = inqry6)'
      addand = "AND"
      addwhere = " "
   end
 ...
if (q_enddate <> "") then
   do
      inqry19 = addwhere || addand || " DATE_TOUCHED <= '" ||
q_enddate || "'"
      'set global (inqry19 = inqry19)'
      addand = "AND"
      addwhere = " "
   end
/* set order conditions */
'set global (inqry20 = " ORDER BY TASK_ID")'
/* call query with global variables */
'run query generic'
```

Figure 4. TASK_QUERY Procedure

TSK ID	TASK NAME	PHASE	DESCRIPTION	PSTART	PEND	PLAN DATE DURATION
1	MATH FUNC	DESIGN	INCLUDE CALC	05-06-1992	05-30-1992	24 ->
2	MATH FUNC	CODE	INCLUDE CALC	06-01-1992	06-30-1992	29 ->
3	MATH FUNC	CODE	INCLUDE CALC	06-15-1992	06-15-1992	15 ->
4	MATH FUNC	TEST	INCLUDE CALC	07-01-1992	07-20-1992	19 ->
5	PUBLICATI	WRITING	BOOKS 1 AND	06-01-1992	06-30-1992	29 ->

Figure 5. Query Report

Figure 5 shows a sample report generated by the application. Note that the report takes advantage of Database Manager's date and time arithmetic functions to include the duration in days next to each date pair. Columns can be removed from the query by deleting them from the appropriate inqry string or the application could use a Query Manager form to omit columns and change the field names in the default report. For example, the column "EXPRESSION 7" (for DAYS(PEND) - DAYS(PSTART)) could be changed to "Plan Date Duration".

CONCLUSION

This application can be extended to include:

- Lookup fields on all of the panels. This would allow standard abbreviations to be used in fields, such as the task phase. (Using abbreviations reduces the amount of space a row will take, which provides a substantial savings for large tables.) For example, once a user has entered an abbreviation into the phase field, he or she could select a lookup action, and a predefined, output-only field next to the phase field would be filled in with the complete phase name. Additional consistency could be facilitated with the addition of a TASK_PHASE table, containing all of the allowable phase abbreviations and full names, along with a primary key on the abbreviation to ensure uniqueness. This also would allow the TASK table to have a foreign key on the phase abbreviation, thus allowing only valid phase abbreviations to be entered into the field.
- Additional TASK table columns, such as the names of the people delivering and receiving the task. This would provide more information than the USER_TOUCHED column, since anyone who had a question about the task could talk with either the person delivering the task or the one receiving it. Lookup fields would come in handy here as well. The regular fields could accept the person's userid, and the lookup fields could show the person's entire name and telephone number.
- An instance procedure that automatically fills in a value for a field. This would be useful in the case where a new field is added, the value of which does not always need to be entered by the user. Such a procedure could run, for example, at the beginning of a panel in add or change mode, and provide a default value for a field, as well as allow the user to change it if necessary. A PRODUCT column in the TASK table would be one example where an instance procedure could be helpful.
- Additional "canned" queries. These could be trimmed versions of the GENERIC query, which add additional constraints under the covers. For example, a "completed task" query could: ask the user for the task name, phase, and/or description; construct the query by reusing the inqry variables; add the additional constraint of ensuring that "the actual end date is not null" to the WHERE clause of the query; and run the generic query for the report.

Just as data may not be static, Query Manager's presentation of panel and construction of queries need not be either. The ideas of dynamically posting panels and building queries can be extended beyond task management applications. Whenever an application requires different panels to be displayed based on differences in data rows, this application structure can be applied. For example, an inventory system could require that the wholesale price for a part be entered before the retail price. And whenever an application needs to shield users from SQL statements, a "generic" query panel could be presented, giving the user the chance to place constraints on the report generated, yet allowing the application to take over defaults, like the sequence and ordering of columns.

REFERENCES

The complete database containing the Database Manager and Query Manager objects for this example application can be found on the ESDTOOLS conference disk.

Query Manager Customized Interface, **OS/2 Notebook: The Best of the IBM Personal Systems Developer**. (G362-0003-00).

(All of the following are part of the Extended Services documentation.)

Query Manager User's Guide, (S04G-1010; 04G1010).

Query Manager Exercises, (S05G-1011; 04G1011).

Structured Query Language (SQL) Reference, (S04G-1012; 04G1012).

Guide To Database Manager, (S04G-1013; 04G1013).

ACKNOWLEDGEMENTS

This article would not have been possible without the insight, expertise, and inspiration provided by Allison Hornung, Hutch Milburn, Richard Romanelli, Alejandra Sanchez-Frank, and Anton Versteeg.

Theodore Shrader, *IBM Personal Systems Line of Business, 11400 Burnet Road, Austin, Texas 78758. Mr. Shrader is an associate programmer working on database tools for the Database Manager product. He joined IBM and the database group in June 1989, and has worked in database application development, including Query Manager, since then. He has a BS in computer science from the University of Texas at El Paso (Go Miners!) and is currently pursuing an MS in computer science from the University of Texas at Austin.*

Presentation Manager

A Software Class for Object-Oriented Programming with C and PM

Hans J. Eisenhuth

by Hans J. Eisenhuth

The PrimaryWindow Class is a software class developed by SE International, Inc. that allows the application developer to use the OS/2® 2.0 dialog box editor for the creation of SAA™/CUA™ conforming primary and secondary windows. It simplifies Presentation Manager® (PM) application development under C tremendously, and provides special support for large as well as small PM projects.

During the last few years, a wave of productivity tools for PM and Windows™ application development has flooded the market. We find code generators for C, C++, and all kinds of more or less object-oriented languages. And many tools provide encapsulated development environments (work benches), promising to ease the application development process.

But what is the right tool? The IS Manager has to decide what to use for corporate application development under OS/2 Version 2.0. Important questions to be asked include:

- Does the tool use a standard programming language?
- How good (=maintainable) is the generated code?
- Does it support object-oriented application development?
- Can we satisfy all our requirements with one tool?
- How easy is it to extend an application beyond the capabilities of a tool?
- How safe is the investment?

All of these questions can be answered easily if we look at OS/2 and Presentation Manager together as a standardized application development base. This is because the programming interface of an operating system is designed to deal with all application requirements.

- PM and OS/2 Version 2.0 provide excellent standards for application development such as a good naming convention for variables and a fixed program structure.
- Combining PM together with C allows the implementation of object-oriented and event-driven applications.
- OS/2 provides with IPF (Information Presentation Facility) an excellent object-oriented documentation method.

PM provides a set of basic window classes for object-oriented programming. If we use only these classes, we engage in which is called "native" PM programming and which is not the easiest task on earth (but extremely logical!). However, by using PM dialog boxes and the dialog box editor, we realize that user interface programming becomes easier. Therefore, the more powerful the window classes that we use, the easier the programming becomes.

Unfortunately, dialog boxes have some serious limitations:

- They are not sizeable.
- They have no actionbar.
- They should only be used for SHORT user dialogs (CUA requirement).

This means that the basic PM window classes don't provide easy programming support for primary and secondary SAA/CUA windows with control windows within the client area. So far, the problem must be solved with native PM programming, which definitely requires a lot of PM skills.

THE PRIMARY WINDOW CLASS

The PrimaryWindow Class is a powerful PM window class that is implemented in the same way as all basic PM window classes. Objects of the PrimaryWindow Class break the limits of dialog boxes. The excellent support for cursor navigation of dialog boxes is maintained and the dialog box editor is used to create primary CUA windows with controls in its client area. No additional tools are needed!

Objects of the PrimaryWindow Class are PM standard windows which:

- are sizeable up to the size of the defined dialog box.
- do not display undefined window areas when the window is maximized.
- contain automatic horizontal and vertical scrolling support.
- support dialog templates which are larger than the physical screen (the primary window is limited to the full screen).
- have an actionbar with mnemonic and accelerator support.
- provide additional functions for keeping the keyboard focus window visible in the client area.
- minimize within a container window and not within the desktop window.
- have automatic support for multiple IPF help libraries (an important requirement for large object-oriented applications).
- have a simple and straightforward programming interface.

Programmers will become more productive, because they now can focus exclusively on the application logic. The application logic for a primary window is coded in the 'case'-statements for:

- Menu selections (WM_COMMAND)
- Control manipulations and data entry (WM_CONTROL)
- Dynamic Data Exchange (DDE messages)
- Direct manipulation (DM_* messages).

All other window behavior is inherited from the PrimaryWindow Class. The PrimaryWindow Class also provides additional methods that simplify the PM programming even more. All methods of the class have the prefix Sei.

CREATING PRIMARY WINDOWS

Only four steps are needed to create a primary window. They are very similar to the creation of a dialog box:

1. Define a dialog template with the dialog box editor (Figure 1).

```
DLGTEMPLATE DLG_MAIN LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "PrimaryWindow Object", DLG_MAIN, 31, 16, 260, 173,
            FS_NOBYTEALIGN | NOT FS_DLGBORDER | FS_SIZEBORDER, FCF_SYSMENU |
            FCF_TITLEBAR | FCF_MINBUTTON | FCF_MAXBUTTON | FCF_VERTSCROLL |
            FCF_HORZSCROLL
    BEGIN
```

Figure 1. Resource Definitions for a Primary Window (Continued)


```
        LTEXT           "~Name", -1, 32, 137, 28, 8, DT_MNEMONIC
        ENTRYFIELD      "Charley Brown", EF_NAME, 82, 139, 135, 8, NOT
                        ES_AUTOSCROLL | ES_MARGIN
        LTEXT           "~Street", -1, 32, 125, 36, 8, DT_MNEMONIC
        ENTRYFIELD      "621 NW 53rd Street", EF_STREET, 82, 127, 135, 8,
                        NOT ES_AUTOSCROLL | ES_MARGIN
        . . . . . .
    END
END

ICON DLG_MAIN main.ico

MENU DLG_MAIN
BEGIN
   SUBMENU  "~File", MN_FILE
   BEGIN
     MENUITEM "~New"            , MN_FILENEW
     MENUITEM "~Open reduced"   , MN_FILEOPEN
     MENUITEM SEPARATOR
     MENUITEM "~Exit \tF3"      , MN_FILEEXIT
   END
   . . . . .
END

ACCELTABLE DLG_MAIN PRELOAD
BEGIN
    VK_F3,       MN_FILEEXIT,   VIRTUALKEY
    VK_NEWLINE,  DID_OK,        VIRTUALKEY
END
```

Figure 1. Resource Definitions for a Primary Window

2. Define a MENU and/or ACCELTABLE and/or ICON resource. The resource IDs must correspond with the dialog template ID (Figure 1).

3. Define the application-specific behavior in a primary window procedure (Figure 2).

4. Create the primary window by invoking **SeiCreateDialogWindow** (Figure 3).

If required, additional methods of the PrimaryWindow Class (e.g. keyboard focus tracking, drag/drop and help support) can be included to extend the behavior of primary windows.

The automatic tracking of the keyboard focus control within the client area is achieved by including the following code in the primary window procedure:

```
case WM_CONTROL: {
      if(SeiIsSetFocusNotification (
                hwnd,
                SHORT1FROMMP (mp1),
                  SHORT2FROMMP (mp1))) {
        SeiMoveToTop (hwnd,
                  SHORT1FROMMP (mp1),
                  FALSE);
        SeiMoveToLeft (hwnd,
                  SHORT1FROMMP (mp1),
                        FALSE);
      }
```

As you will see later, it is very likely and very important to have multiple help libraries within large PM applications. By adding the following code template to a primary window


```
MRESULT EXPENTRY DlgWinProc( HWND hwnd, USHORT msg,
                             MPARAM mp1, MPARAM mp2 )  {
  switch (msg) {
    case WM_INITDLG: {
         WinSetDlgItemText (hwnd, EF_NAME, PVOIDFROMMP (mp2));
         } break;

    case WM_COMMAND: {
         HWND   hwndDummy;
         switch (SHORT1FROMMP (mp1)){
           case MN_FILEOPEN:
                /*-------------------------------------------------------*/
                /* Create a new primary window for the current container*/
                /*-------------------------------------------------------*/
                hwndDummy = CreateDummy (SeiQueryContainer (hwnd),
                                         "Little John Doe");
                break;

           case MN_FILEEXIT:
                WinDestroyWindow (hwnd);
                break;

           default:
                DosBeep (500,100);
                break;
         } /* end WM_COMMAND switch */

         } return (0); /* !!! Very, very, very IMPORTANT !!!!!     */
                       /* Never use break at the end of WM_COMMAND */

    default:
         break;
    } /* end switch */
  /*-----------------------------------------------------------*/
  /* Don't call WinDefDlgProc BUT SeiDefDlgProc !!!!!!!!      */
  /*-----------------------------------------------------------*/
  return((MRESULT)SeiDefDlgProc(hwnd,msg,mp1,mp2));
}/* end of DlgWinProc */
```

Figure 2. Primary Window Procedure

procedure, the activation of another help library is greatly facilitated.

```
case WM_ACTIVATE: {
       SeiChangeHelpTable (
           SeiQueryDialogFrame (hwnd),
              DLL_NAME,
              DUMMY_HELPTABLE,
              HELP_LIB,
              SHORT1FROMMP (mp1));
       } break;
```

Figure 4 shows a primary window. During the creation of the window, we specified a container handle. It is the owner of the primary window. This window handle has a certain impact on the behavior of the primary window:

- If the container window is minimized, all primary windows disappear from the screen.


```
HWND EXPENTRY CreateDummy (HWND hwndContainer, PARAMDUMMY *pParamDummy )
{
usrc = DosGetModHandle (DLL_NAME, &hMod);

hwndFrame = SeiCreateDialogWindow (HWND_DESKTOP,        /* Parent
*/
                                   hwndContainer,       /* Owner
*/
                                   WS_VISIBLE,          /* Frame style
*/
                                   NULL,                /* FCF
*/
                                   DlgWinProc,          /* WinProc
*/
                                   NULL,                /* Title
*/
```

Figure 3. Constructor of a Primary Window

- If a primary window is minimized, its icon appears in the client area of the container. This default behavior can be changed.

For proper user interface programming it is very important to separate the code to access DBMS from the PM window procedures by implementing object-oriented data base access classes. This separation facilitates the maintenance of the entire application. Should new data access methods be used in the future, the investment in the user interface development is secure.

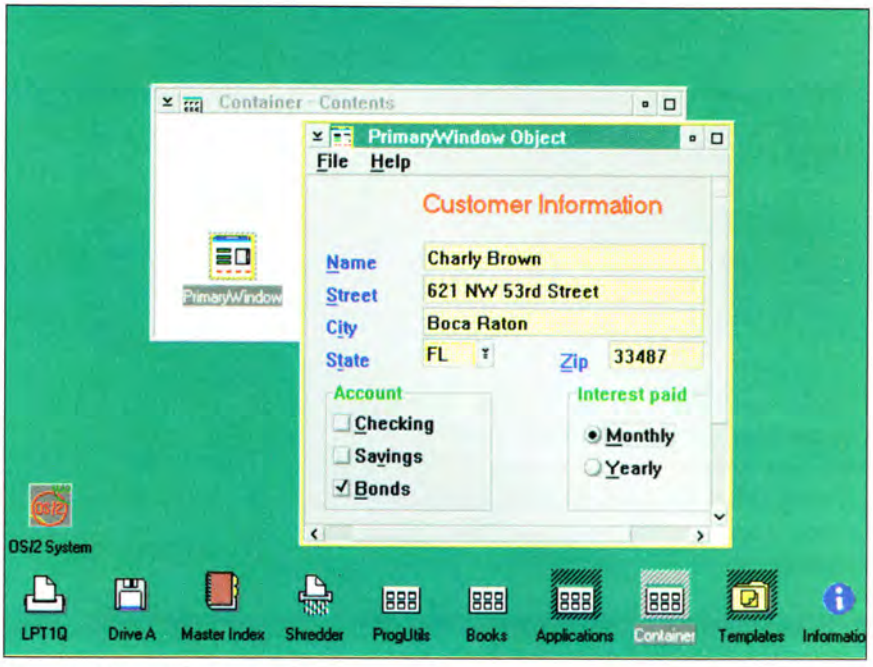


Figure 4. Primary Window Sample

DOCUMENTING WINDOW CLASSES

Today each class library vendor has its own more or less object-oriented documentation style, and developers using these libraries have to become familiar with each one. However, documentation should be standardized, and fully object-oriented.

The IBM® OS/2 Programming Reference in Version 2.0 (and in Version 1.x) offers an excellent and complete object-oriented documentation style for classes. Figure 5 shows the contents window of the Programming Reference.

Each basic PM class is documented in the same way by:

- objects styles.
- parameters (control data) that must be provided to create an object.
- reactions of objects to important events (notifications).
- applicable methods (messages or functions).

The Programming Reference is the developer's most important programming tool. Therefore, it makes sense to use IPF to document (window) classes according to this standard, and to incorporate the documentation into the OS/2 Programming Reference.

The great benefit of PM applications is that they have a simple and re-usable program structure, which is independent from the application complexity. This is actually the reason that we see so many code generators for Windows and PM applications. Having understood the structure and philosophy of PM applications, it makes perfect sense to include even reusable code templates into the Programming Reference. These code templates can be added easily to the application code via the clipboard.

Hence, the IBM OS/2 Programming Reference is a perfect solution for a standardized and object-oriented documentation tool. And everybody who writes OS/2 window classes or class libraries would do well to follow this example.

DEVELOPING LARGE PM APPLICATIONS

Organizing the workload among different development groups is the most important task when developing large applications. In an object-oriented development environment, groups should develop independent and re-usable classes like the PrimaryWindow Class that have no side effects on other classes. These classes should be implemented as dynamic link libraries (DLLs), whereby the dynamic linking reduces the overall maintenance efforts.

Figure 6 shows the development environment of a (window) class X. Each development group should follow this scenario. The following files are "exported" because they must be accessible by other applications or objects:

- **X.H:** Header file which contains external function prototypes, user messages, and type definitions for control data and other external data structures.

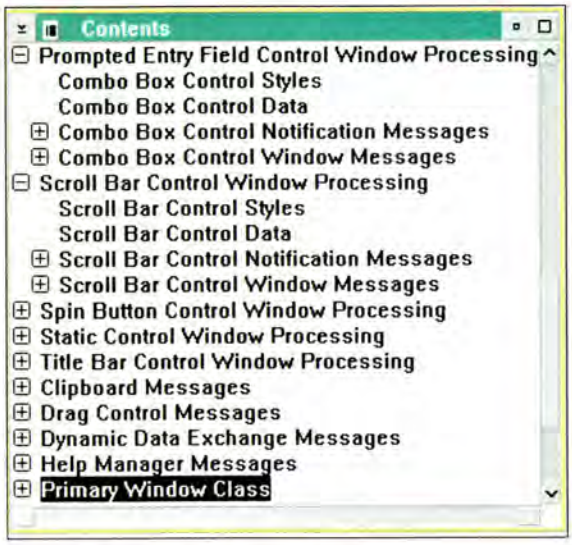


Figure 5. Programming Reference Structure

- **X.DLL:** Dynamic Link Library which contains the code for object constructors and window procedures.

- **X.LIB:** Import library to resolve external references to X during the link process with other objects.

- **X.HLP:** Help library which contains the help text for contextual and extended help support.

- **X.INF:** Information file which contains the online documentation that should be included in the IBM OS/2 Programming Reference.

In larger PM and C projects we applied this scenario and experienced that:

- by using code templates and a fixed file structure, the communication among programmers of different programming groups improved substantially.

- for quality assurance, it was much easier to detect "programming patterns" that do not follow the standards of PM application development.

- the maintenance effort was reduced due to the fixed structure of PM window procedures.

- programmers start to develop useful and reusable window classes, and these classes become an important asset of programming departments.

In addition, we recognized a drastic reduction in learning time for PM programmers. By providing the PrimaryWindow Class *and* a training in object-oriented application development with PM and C, programmers immediately were able to write rather sophisticated workplace applications. Later they learned the details of Presentation Manager programming while already writing CUA-conforming standard applications.

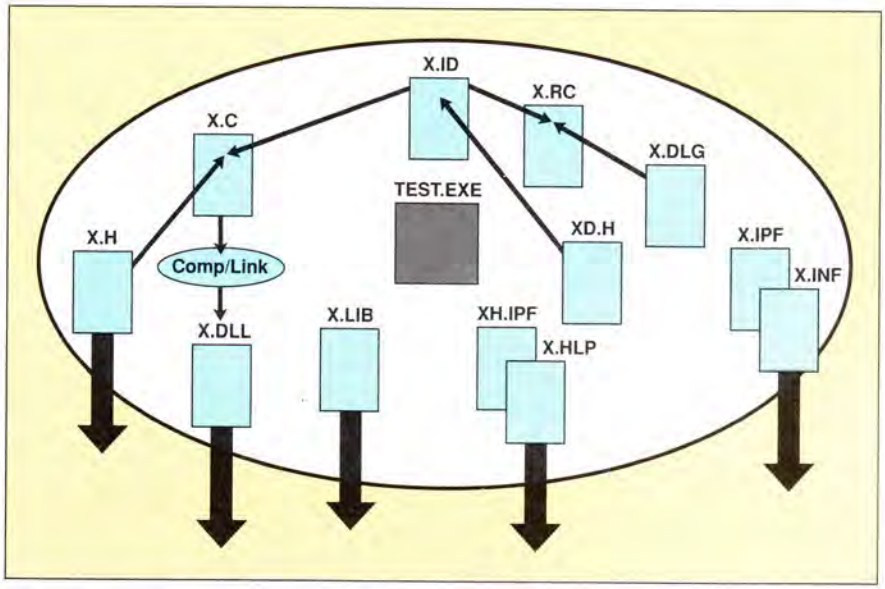


Figure 6. Development Environment of a Class X

No matter which development tool you use, the understanding of the impact of the SAA/CUA object-action approach on user interface design will always be more difficult than the actual coding. Therefore, using powerful software classes written at an operating system level enables high productivity from the very beginning.

By using software classes developed with PM and C you will benefit from further advantages:

- Independence from a specific tool vendor (only OS/2 is the standard).
- No limitations on the functionality of a class.

SUMMARY

OS/2 provides many characteristics of an object-oriented operating system. Through the use of the programming standards of the OS/2 programming interface, and the fixed structure of Presentation Manager applications and the OS/2 Programming Reference as an object-oriented documentation tool, we are able to develop truly object-oriented applications and manage large object-oriented projects.

The PrimaryWindow Class demonstrates that through the use of powerful software objects the overall program complexity is reduced. The PrimaryWindow Class also is a good example how to apply OS/2 standards for object-oriented application development.

As soon as we see more software classes like the PrimaryWindow Class, based on nothing else but OS/2 Version 2.0 and Presentation Manager, this exciting operating system will become the standard for object-oriented and multitasking applications.

Let's do it !

Hans J. Eisenhuth, *SE International, One Park Place, Suite 240, 621 NW 53rd Street, Boca Raton, FL 33487, (407)241-3428, Fax: (407) 997-8945. Mr. Eisenhuth is founder and president of SE International, Inc., and SES GmbH, Berlin, Germany. Both companies focus on consulting and education in object-oriented application development and user interface design under OS/2. Mr. Eisenhuth holds the equivalence of a BS in Mathematics and Economics and an MS in Computer Science from the Technical University Berlin.*

A demo version of the PrimaryWindow Class can be ordered from SE International, Inc., or downloaded from the IBMOS2 library on CompuServe.®

Presentation Manager

Programming Printing Under OS/2

by David E. Reich

*In the Summer 1991 issue of the "IBM® **Personal Systems Developer**," we discussed how to use the printing functions provided by OS/2.® We also discussed the overall subsystem architecture. In this issue we will explore how to write programs that print and discuss the advantages and drawbacks of the various functions you may choose to support.*

All of the information presented here is applicable to OS/2 version 2.0 as well as version 1.3. The basic architecture is not changing and by following the recommendations herein, you will be able to get your programs running in both environments with ease.

Throughout the discussion in this article, we will refer to the OS2.INI and OS2SYS.INI files as one, called the INI file. By using the Prf and DosPrint API calls, there is no need for application programs to know the difference. For completeness, the OS2.INI file stores information about programs and groups, while the OS2SYS.INI file stores information about printers and queues. The APIs make this difference transparent.

As you will see, the INI file is really the central point in programming your applications to print. The INI file contains a wealth of information, and as long as an application pulls this information out, very little else needs to be done to enable it to print.

Note that throughout this article, you will see how you are "supposed" to do things in applications. These are not hard and fast rules; they are recommendations on how to do things to make life easier for the application user. You are free to use these suggestions as you wish. Also please note that only program segments will be included here. As you will see, these functions need to be executed in different areas of a program. You are free to implement them as you prefer. Only the functions need to be executed; the method of presenting user choices (listboxes, checkboxes, etc.) is unimportant. The important functions shown here can be pasted directly into a program in whatever manner you desire.

David E. Reich

WHERE TO START

I have found that the best way to start looking at this, both in terms of understanding things, and in how to use them practically, is to go backwards. Sometimes it is more informative to think about what you want, then find out what elements it is comprised of, then what they are comprised of, and so on. This is definitely one of those times.

Device Context

In order to display anything under the OS/2 Presentation Manager,® you need a device context (DC). For the purposes of this article, we are dealing with printer device contexts. A printer DC is obtained by a call to DevOpenDC. In the DevOpenDC call, you must specify the type of device context (metafile, memory, info, or for printing, queued or direct.) It is highly recommended to use only queued (OD_QUEUED) DC's for printing. This simplifies the printing functions on the programmer's part and also allows maximum throughput for the application and the system on which it is running.

Additionally, as you will see shortly, the recommended print destination is a queue.

Back to the DevOpenDC call - A DEVOPENSTRUC is passed to indicate what device the device context is for. The most important piece of information needed in a DEVOPENSTRUC is a DRIVDATA structure. This is the information that will be used to create the DC specific to the printer and queue destination. The DEVOPENSTRUC and DRIVDATA structures are defined in file OS2DEF.H in the IBM Toolkit and in the IBM Presentation Manager Programming Reference, Vol. 1. The importance of this structure cannot be overstated. This is what tells the printer driver how to do everything associated with the print job. The DRIVDATA tells what queue, what printer driver and what job properties to associate with the job. The following discussion and code will show you how to fill in this structure with a minimum of work while giving users maximum flexibility in how they can create print jobs.

Print Destinations

A print destination can be a queue, a printer or a port. A port destination is specified as LPT1, for example. A printer destination is specified by the printer name, such as PSCRIPT1. It is most desirable to use queues as print destinations. Again, this allows maximum flexibility for the user and a minimum of effort for the programmer. My recommendation is to use queues exclusively as the print destination. As you will see, this does not diminish the amount of flexibility applications can provide users.

The reason for this recommendation is so that users may configure queues as they wish, possibly associating more than one queue per printer and in some cases, more than one printer per queue.(something known as printer pooling). The user can then simply select the desired queue and not worry about printer names. This will give a consistent look to printing. Writing applications with queues as print destinations also adds some other advantages which you will soon see. We will assume that the print destinations from this point on are queues.

During the course of application execution, the user will ask for the work to be printed. The first order of business is to open the device context. Before a device context for the printer can be opened, the application first must ask the user for the print destination. A list of queues can be obtained by a call to DosPrintQEnum. An example of this code can be found in Figure 1. This is a C language function which actually inserts a list of all available local queue names into a listbox. Note that you may display the names for selection in any way you wish.

The are a couple of things to note about calling DosPrintQEnum. First, at any one time the function can only be called to enumerate the queues on the local workstation or a particular server, (and for the server, you must have the server name). What this means is that you may have to make multiple calls to the API to enumerate all queues available to the user, first for the local workstation queues, and then for each server (determined by network API's). DosPrintQEnum is called to get the queues on each machine.

Another item to note is the size of the buffer to allocate for the queue. You can see in Figure 1 that we used a 32K buffer. It may seem intuitive to call DosPrintQEnum with info level 0, just to get the names and the number of queues available, and then allocate enough memory to hold the info level 3 structure (which will be detailed in a moment) times the number of queues. This will not work. The reason is that the PRQINFO3 structure consists mostly of pointers, pointing to data areas containing the actual information. Allocating memory for the structures will give you only enough room for the pointers. DosPrintQEnum also returns all the data pointed to. Since there is no way to accurately determine the size of the data before the call, the best move is to allocate a large buffer. The way it is done here is to try a 32K buffer, and if that is not enough, allocate 64K. You can tell whether you simply need more buffer area if you get the error code ERROR_MORE_DATA. In that case, reallocate the buffer and call the API again.

Back to the function. When DosPrintQEnum is called, you can request one of 5 information levels, 0 though 4. For the purposes of generating a print job, the best one to use is info level 3. When this level is used, an array of PRQINFO3 structures is returned in a buffer. This is the buffer mentioned earlier.


```
/*****************************************************************************/

VOID QListBxFill( HWND hwndQLB )
{
  SEL           selarQInf3; /* Selector for array of PRQINFO3 structures  */
  USHORT        usRetEnt;   /* Returned number of queue info 3 structures */
  USHORT        usAvailEnt; /* Number of available queues                 */
  USHORT        usStructBufSize;  /* Size of the buffer to allocate       */
  PBYTE         pbuf;       /* Pointer to the buffer of info structures   */
  PBYTE         pbuf1;      /* Pointer for offsetting into structures     */
  USHORT        usErr;      /* Return code                                */
  PPRQINFO3     pPQInfoStruct;  /* Pointer to PRQINFO3 structure          */
  USHORT        ctr;        /* A counter                                  */

  /* First, blast anything in the listbox                                  */
  WinSendDlgItemMsg( hwndQLB, ID_QUELB, LM_DELETEALL, NULL, NULL );


    /* Let's get the queue info 3 structures                */
    /* First, allocate the buffer. Then call DosPrintQEnum. */
    /* If we error because there is more data, allocate a   */
    /* Larger buffer and do it again. If not, count through */
    /* Each entry and insert the queue name in the listbox. */


 /* Allocate space for queue names */
    usStructBufSize=32768;                  /* Try a 32k buffer first  */
    if (!DosAllocSeg(usStructBufSize, &selarQInf3, 0 ))
       {
       pbuf = MAKEP(selarQInf3, 0);
       }
       else
          {
            WinMessageBox( HWND_DESKTOP, NULL,
            "DosAllocSeg Failed for q info structs", "Fail!!!!!", 1, MB_OK );
            }

    /* Enumerate the queues */
     usErr = DosPrintQEnum((PSZ)NULL,3,(PBYTE)pbuf,
                           usStructBufSize,
                           (PUSHORT)&usRetEnt, (PUSHORT)&usAvailEnt);

       if (usErr != ERROR_MORE_DATA)              /* If there's more data, */
          {                                       /* Try again with a      */
            DosFreeSeg(selarQInf3);               /* Bigger buffer         */
            usStructBufSize=65535;                /* Try 64k               */
     if (!DosAllocSeg(usStructBufSize, &selarQInf3, 0 ))
                    {
                    pbuf = MAKEP(selarQInf3, 0);
                    }
                    else
```

Figure 1. Function to Query Queues and Insert Queue Names into a Listbox (Continued)


```
                    {
                      WinMessageBox( HWND_DESKTOP, NULL,
                      "DosAllocSeg Failed ", "Fail!!!!!", 1, MB_OK );
                    }

                  /* Enumerate the queues (Again) */

          usErr = DosPrintQEnum((PSZ)NULL,3,(PBYTE)pbuf,
                                usStructBufSize,
                                (PUSHORT)&usRetEnt,
(PUSHORT)&usAvailEnt);
          }
          else
          {
             WinMessageBox( HWND_DESKTOP, NULL,    /* For any other error
*/
             "DosPrintQEnum Failed",               /* just report it
*/
                           "Getting level 3!", 1, MB_OK );
          }

   if (!usErr)                                     /* If no error, */
      {
       pbuf1=pbuf;
       for (ctr=1;ctr<=usAvailEnt ;ctr++ )
           {
            pPQInfoStruct=(PPRQINFO3)pbuf1;       /* Fill the listbox */
            WinSendDlgItemMsg( hwndQLB,
                               ID_QUELB,
                               LM_INSERTITEM,
                               MPFROM2SHORT( LIT_SORTASCENDING, 0),
                               MPFROMP(pPQInfoStruct->pszName));

            pbuf1=pbuf1+sizeof(PRQINFO3);         /* Offset by another
*/
           }                                      /* PRQINFO3 structure
*/
```

Figure 1. Function to Query Queues and Insert Queue Names into a Listbox

The PRQINFO3 structure is defined in the PMSPL.H file that is included with the IBM Toolkit, and can be found in the IBM Presentation Manager Programming Reference, Vol 1. Recall that the structure has pointers for most of its elements. The data also are returned from the call, after the structures. Figure 2 shows an example of the buffer format. This knowledge is not critical in programming the function, but it is included here for completeness and help in application debugging.

For simplicity in this programming example, the assumption is made that only local queues are of interest. Therefore there is only one buffer to deal with, as in the example in Figure 1. We are still at the point where the user has selected a menu item which indicates to the application that he or she would like to print.

A call to DosPrintQEnum has returned the array of PRQINFO3 structures, so all available queues are known. The next thing to do is to ask which queue the user would like to print to. Referring to Figure 1, a listbox gets filled with the queue names. Optionally, you may wish to include the queue description, also

part of the level 3 structure, as part of the listbox entry. Again, you may implement queue selection in a different way, if you prefer.

Now, the user has selected a queue. A device context is still needed, and only some of the information needed is now available. Returning to the premise of working backwards, the item that is most important in the creation of a printer device context is the DRIVDATA structure. This contains all of the job properties and other information for the print job. Selecting the queue is the first step in getting this structure.

There is a large amount of flexibility in how to specify job properties for print job. This has led to some confusion among developers. The following is the recommended way to work with properties.

Recall from the last issue of the Developer, that there are job properties which are stored as default job properties, and are used if a job comes in without any of its own. These should not be used by Presentation Manager programs. Think of them as for use by base, or command line printing, such as "COPY CONFIG.SYS LPT1:". The job properties that are stored on a per-queue basis are the ones that PM programs should be using.

A DRIVDATA structure is an element of the PRQINFO3 structure, returned from DosPrintQEnum. This is a preformatted structure that can be used directly in the DevOpenDC call, as the DRIVERDATA part of the DEVOPENSTRUC. This contains the job properties for the queue as stored in the INI file. These setting are stored by the Print Manager program.

You may wish, however, to give the user the chance to modify the properties stored with the queue. The way to accomplish this is with DevPostDeviceModes. This is another API which, when called, tells a printer driver to put up a dialog.

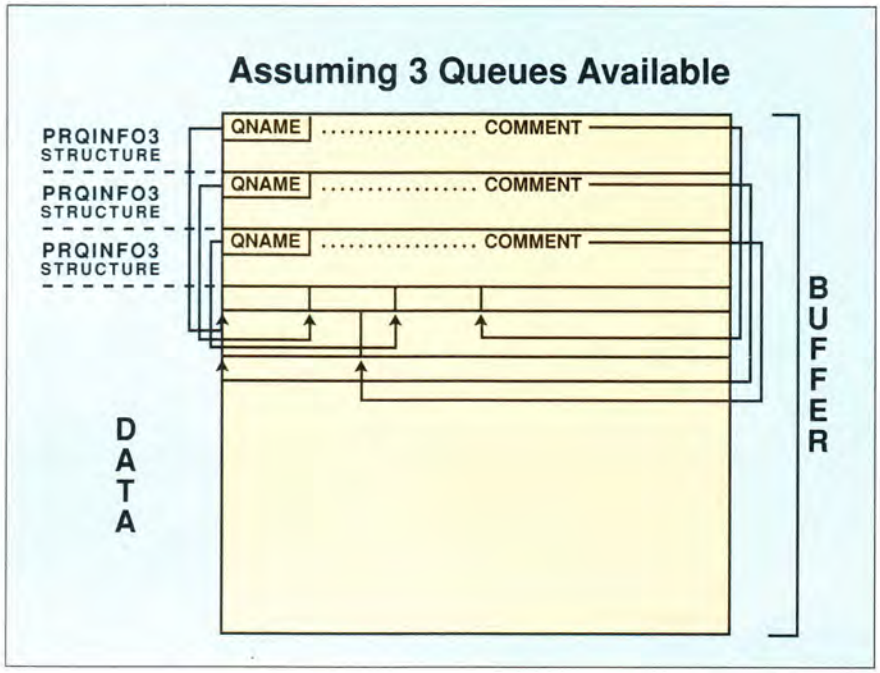


Figure 2. Buffer Structure Returned from DosPrintQEnum (Info level 3)...

```
/*************************************************************/
/* Call DevPostDeviceModes to get the properties for the job */
/* before calling DevOpenDC                                  */
/*************************************************************/

   length = DevPostDeviceModes( habPrn, NULL, szDDName,
                                szDevName, szPrinter,
                                DPDM_QUERYJOBPROP);

     if (!DosAllocSeg(length+4096, &selpDrivData, 0 ))
        {
          pDrivData = MAKEP(selpDrivData, 0);
        }
        else
         {
```

Figure 3. Calling DevPostDeviceModes (Continued)


```
        WinMessageBox( HWND_DESKTOP, NULL,
        "DosAllocSeg Failed ", "Fail!!!!!", 1, MB_OK );
       }

/*************************************************************/
/* Call DevPostDeviceModes to put up the job properties      */
/* dialog so that user can choose to change from defaults    */
/*************************************************************/
 rc = DevPostDeviceModes( habPrn, pDrivData, szDDName,
                          szDevName, szPrinter, DPDM_POSTJOBPROP);
```

Figure 3. Calling DevPostDeviceModes

The way to do this is shown in Figure 3. A call to DevPostDeviceModes, specifying the queue name selected previously, will tell the printer driver associated with that queue to put up a dialog. The example in Figure 3 first calls the function only to query the job properties. This returns the size of the driver data structure. The main reason for this call is to obtain the size of the data structure needed for possible modification. Next is a memory allocation for a buffer size slightly larger than this (dependent on the size of the modifications). In this example we used 4096 bytes larger than the present size.

Next comes the call to DevPostDeviceModes specifying DPDM_POSTJOBPROP. This will put up the job properties dialog with the queue-specific defaults filled in. The user can then change any of those properties. Upon return, the DRIVDATA structure is filled in with the user's choices.

Where are we so far? Well, the queues have been queried and the user has selected one. From that, the user also has had the opportunity to change any of the job properties associated with that queue. The DRIVDATA structure is complete. Now we can create the device context.

Figure 4 shows the syntax of the DevOpenDC call. It is very straightforward. Using the DRIVDATA structure from DosPrintQEnum along with the other information returned from DosPrintQEnum, an entire DEVOPENSTRUC can be created easily. This illustrates what was mentioned earlier about pulling the information from the INI file. The other items specify how the job is to be printed, such as PM_Q_STD or PM_Q_RAW.

Now, the device context has been created; the first half of the work is done. The next step is to create the presentation space and draw into it. OS/2 does the rest.

```
/*********************************************************************/
/* Fill in the DEVOPENSTRUC. Use the selections and the DRIVDATA     */
/* obtained earlier from DosPrintQEnum and DevPostDeviceModes        */
/*********************************************************************/
DevData.pszDriverName=pszDDName; /* Device driver name from selected queue */
DevData.pdriv=pDrivData;         /* DRIVDATA structure                     */
DevData.pszDataType="PM_Q_STD";  /* Specify standard printing (not raw)    */
DevData.pszComment="TestPrint";  /* Print Job Comment                      */
DevData.pszQueueProcName=NULL;   /* Take queue processor default           */
DevData.pszQueueProcParams="XFM=0";  /* Default Parms                      */
DevData.pszSpoolerParams=NULL;   /* No Special Spooler or network parms    */
DevData.pszNetworkParams=NULL;
DevData.pszLogAddress=szQueName; /* queue name selected from listbox */
```

Figure 4. DevOpenDC Call (Continued)


```
/*******************************************************/
/* Get a device context for the queue                  */
/*******************************************************/
hdcPrn = DevOpenDC( hab,                        /* anchor block handle */
                    OD_QUEUED,                  /* queued printing     */
                    "*",                        /* default token       */
                    9L,                         /* 9 items in DevData  */
                    (PDEVOPENDATA)&DevData,     /* pointer to DevData  */
                    (HDC)NULL );                /* no compatible DC    */

if( hdcPrn == DEV_ERROR )
{
  ErrorCode = WinGetLastError( hab );
  ErrorOccurred = TRUE;

  WinMessageBox( HWND_DESKTOP, NULL,
    "DevOpenDC Failed", "Fail!!!!!", 1, MB_OK );

}
```

Figure 4. DevOpenDC Call

The Presentation Space

Once you have a device context (DC), a presentation space (PS) must be created. This is the device-independent entity that allows the application to draw without regard to the output medium. When the PS is associated with a DC, the translation takes place at the association link to take advantage of the capabilities of the output device. Figure 5 shows the creation of a presentation space associated with the device context created earlier.

As you can see, this is a very generic call. This goes along with the fact that a PS is a device-independent entity.

There are some cases where applications may wish to know something about the device being printed to. For example, a page oriented application such as a presentation graphics program or page layout program may want to know how big a page is, in order to format the screen properly. A word processor may want to know what fonts are resident on the device. This is not unusual, nor does it defeat device

```
/*******************************************************/
/*        Get a PS for a default page size             */
/*    and implicitly associate it with DC from before  */
/*******************************************************/
szl.cx = 0; szl.cy = 0;

hpsPrn = GpiCreatePS( hab, hdcPrn, &szl,
                      PU_ARBITRARY | GPIA_ASSOC );

if( hpsPrn == GPI_ERROR )
{
  ErrorCode = WinGetLastError( hab );
  ErrorOccurred = TRUE;

  WinMessageBox( HWND_DESKTOP, NULL,
    "GpiCreatePS Failed", "Fail!!!!!", 1, MB_OK );

}
```

Figure 5. Creating the Presentation Space

independence. They still need know nothing about the device to make maximum use of its resolution, for example.

The way to query a page size is to use the function DevQueryHardCopyCaps. This function will return the capabilities of the device. An example of this is in Figure 6. The function, just like the others before, is called twice, the first time to determine how many different forms are available, and the second time to query a specific form. One of the parameters to this API is the handle to the device context obtained earlier. This way the API knows which device to query.

Once the dimensions are known, the program can proceed to format each page and send it off to the printer. Figure 7 shows a pseudo-code example of this. The sequence is initiated with a call to DevEscape. To initiate a document, the escape is DEVESC_STARTDOC. If the document is only one page, the next DevEscape will be DEVESC_ENDDOC. That signals to the printer driver that the document is finished. If it is a multiple-page document, there may be one or more DEVESC_NEWFRAME escapes in between.

```
/*********************************************************/
/* Query hard copy capabilities of printer               */
/*********************************************************/

NumForms = DevQueryHardcopyCaps( hdcPrn, 0L, 0L,(PHCINFO)NULL );

pHcInfo = (PHCINFO)malloc((UINT)(FormsReturned*sizeof(HCINFO)));
   if (!DosAllocSeg(NumForms*sizeof(HCINFO), &selpHcInfo,0 ))
      {
        pHcInfo = MAKEP(selpHcInfo, 0);
      }
      else
       {
        WinMessageBox( HWND_DESKTOP, NULL,
        "DosAllocSeg Failed ", "Fail!!!!!", 1, MB_OK );
       }


DevQueryHardcopyCaps( hdcPrn, 0L, NumForms, pHcInfo );

us=0;
                /* Cycle through to get the current caps */
while( pHcInfo[us].flAttributes != HCAPS_CURRENT)
     { us++; }

/*********************************************************/
/* Set the height and width of the printer's page size */
/* This will be used later by the function that draws  */
/* into the PS                                           */
/*********************************************************/

cyPage = pHcInfo[us].yPels;
cxPage = pHcInfo[us].xPels;
```

Figure 6. DevQueryHardCopyCaps Example


```
/*********************************************************/
/*            Initiate the document                      */
/*********************************************************/


DevEscape( hdcPrn, DEVESC_STARTDOC, 8L,
           (PBYTE)"Sample\0", (LONG)0, (PBYTE)NULL );


Draw Stuff........
Draw more stuff........
(Now at end of a page,)


DevEscape( hdcPrn, DEVESC_NEWFRAME, 0L, NULL, 0L, NULL);


 Now draw on the next page.......
 And continue, using the above 2 statements until there's nothing left
 to draw......



/*********************************************************/
/*            Terminate the document                     */
/*       The ID of the job is returned to us             */
/*********************************************************/

 OutData=2;                              /* Required for an ENDDOC */
 DevEscape( hdcPrn, DEVESC_ENDDOC,
            (LONG)0, (PBYTE)NULL,
            &OutData, (PBYTE)&usJobId );
```

Figure 7. DevEscape Printing Sequence

Once the job is finished and the DEVESC_ENDDOC is sent, the PS and DC should be destroyed. Then, any memory and structures allocated for the printing function should be cleaned up as well.

The sequence of events then, is as follows:

1. The user wishes to print and selects the "Print" menu item in the application.

2. The application uses DosPrintQEnum to present the user with a list of available queues. The user selects one.

3. Next, the job properties must be determined. This is done by calling DevPostDeviceModes to put up the Job Properties dialog, which allows the user to see the properties for the queue and, optionally, to change some.

4. The device context is created with DevOpenDC.

5. A presentation space is created with GpiCreatePS and associated with the DC.

6. Using DevEscape, a DEVESC_STARTDOC is sent followed by commands to draw into the PS. Possibly some DEVESC_NEWFRAME escapes are sent in between to indicate page breaks. The DEVESC_ENDDOC then is sent to indicate the end of the job.

7. Data areas, DC handles and PS handles are cleaned up and deallocated.

That really is all there is to coding printing. Of course, this is a simplified example, but it shows the skeletal structure of how to code printing and which path you should take with respect to sending jobs to ports, printers, or queues. Just like any output under the OS/2 Presentation Manager, all that are needed are a device context and a presentation space. The process of obtaining these items is what can confuse things. There are many ways to get the information you will need.

OTHER CONSIDERATIONS

Threads

Your code should be structured such that the printing is done on a separate thread from the main message processing thread. You should be sure to provide the user with a button to cancel the print job. In this case, the application should send a DEVESC_ABORTDOC. This will abort the job being sent.

By placing your printing code on a separate thread, you allow your program which currently has the focus to be responsive to the message queue, affording the user the opportunity to switch to another program while yours is printing. There is something to watch for here. That is to be sure you do not allow the user to change the display of your application while the printing is in progress. It does not make sense to begin a print job and then allow the user to change what is being printed after the job has started but before it is completely spooled.

Flexibility

The OS/2 Print Subsystem is very flexible. It allows a user to configure many queues per printer driver, and many printers per queue. There are several ways to obtain the DRIVDATA structure needed for DevOpenDC. Queues are the most flexible and provide the most information with the least amount of code. By querying the queues with DosPrintQEnum, an application will have all of the information associated with that queue, including all of the printer drivers it is connected to. In the previous examples, we used the default printer for the queue, but is a trivial matter to use a different printer.

When the user selects a queue, the printer names are part of the PRQINFO3 structure. These can be pulled out and put in a selection list of their own, so the user can select a queue with a printer other than the default. In this way, the user can use one type of destination, the queue, to print all jobs. Following the recommendation of using only queues as destinations makes learning easy for the user, as he or she only needs to know about pointing to a queue. Once a queue is selected, you have the option of allowing the user to specify another printer. Something to note from experience is that in most cases, only one printer is associated with a queue, so the situation becomes even more straightforward.

Another point about using queues is that when users look at the Print Manager window, what they see are queues. In a large number of installations, users know nothing about printer names versus port names or queue names. All they know is what they see in the Print Manager, which is a list of queues. They know that they want to print to this queue or that queue, so using queues becomes even more intuitive to the user. Again, you may choose to add additional choices for the user as you wish.

Finally, any discussion about printing would not be complete without mentioning fonts. There is a companion article in this issue that deals with the issue of programming fonts under OS/2.

SUMMARY

On the surface, writing a program that prints under OS/2 looks more involved than it is. Once you understand the basics of the architecture and the concepts presented here, it becomes a simple task.

This article eliminates the confusion concerning how to program for printing. Since there is so much flexibility and information available, many people have been unsure about how to write their code. By using the information in the INI file and following these suggestions, you will provide your application users with the flexibility and ease of use inherent in OS/2.

David E. Reich, *IBM Corp, Internal Zip 1424, 1000 NW 51st Street, Boca Raton, Florida 33429, is a senior associate programmer working with OS/2 Technical Support in Boca Raton. He has an MS in Computer Science from the State University of New York at Albany, has written several articles previously in "IBM Personal Systems Developer", and has co-authored a book entitled OS/2 Presentation Manager Programming, published by John Wiley & Sons, Inc.*

REFERENCES

IBM OS/2 Presentation Manager Programming Reference, Vol. 1 (S10G-2625).

Cheatham Paul W., David E. Reich and Robert F.G. Robinson. **OS/2 Presentation Manager Programming**, John Wiley & Sons, Inc. (ISBN 0-471-50897-7).

Reich, David E. **Printing Using OS/2**, *IBM Personal Systems Developer*, page 101, Fall 1991.

Reich, David E. **The OS/2 Print Subsytem Architecture**, *IBM Personal Systems Developer*, page 107, Fall 1991.

Presentation Manager

Programming with Fonts Under OS/2

David E. Reich

by David E. Reich

Although the catch phrase during the last few years has been, "The Paperless Office", hard copy output has become more important than ever before. There are new printers being introduced all the time, along with new ways of displaying characters and graphics on them.

In late 1990, IBM® introduced the Adobe Type Manager®, or ATM, font technology into the OS/2® 1.3 product. This opened up a whole new world of typefaces and WYSIWIG (What You See Is What You Get) function to the OS/2 product. This technology provides fast, high quality fonts to OS/2 programs for both screen and hardcopy output. There are now several font "types" available in OS/2, ranging from bitmapped fonts to these ATM fonts. This article will define the differences and show you how to request the various types of fonts.

Also note that the information presented here, as with the other articles in this series on printing, is applicable to OS/2 version 1.3 as well as version 2.0. Any differences are noted in the text.

You will notice that throughout this article, you will see how you are "supposed" to do things. These are not rules. They are recommendations on how to select fonts to provide the highest quality output for your users. You are free to use these suggestions as you wish. Also please note that only program segments will be included here. As you will see, these functions need to be executed in different areas of a program. Again, you are free to implement them as you like. Only the functions need to be executed; the method of presenting user choices (listboxes, checkboxes, etc.) is unimportant. The important functions shown here can be pasted directly into a program in any way that you find convenient.

SCREEN VS. PRINTER FONTS

When one speaks about fonts in OS/2, he or she may be referring to printer fonts or screen fonts. The two are distinct, and to make the best use of the OS/2 font technology, any font installed for the printer should also be installed for the screen. This is not a requirement, but for best results, both should be installed.

Screen fonts are those that can be displayed on the screen in Presentation Manager® applications. OS/2 comes with a set of screen fonts that are installed at the user's request during system installation. Others come on separate diskettes and are available from IBM and other vendors. Some applications such as Aldus®' Pagemaker® come with a set of screen fonts.

Once a font is installed, it is called a public font and is available to all applications. That is, if an application queries the screen fonts available, all of the fonts that have been installed will be returned.

Printer fonts are those installed for a particular printer driver. Some printer drivers support downloadable fonts. PostScript® printers are such an example. In this case, the printer fonts must be installed into the printer driver to be downloaded. As you will soon see, this makes the downloading operation transparent to the application program.

An item to note about printer fonts is a recent enhancement to the IBM 4019 and HP® Laserjet® drivers. Previously, if a non-printer font was to be used, each bitmap would be

sent to the printer. With this new enhancement, a bitmap is sent to the printer, but it is then stored as a printer font character. As such, it subsequently can be manipulated as a device font.

FONT TYPES

Before any discussion of architecture can take place, you must understand the different types of fonts available in OS/2.

Bitmap

The simplest type of font available is the fastest, but least flexible. It is the bitmap font. Some have opted to call it an "image" font, as well.

A bitmap font is exactly that: a series of maps of bits, stored in a file. The map has each pel within it turned on or off corresponding to how the character being represented is supposed to look. Each set of bitmaps corresponds to the character set at a particular point size (unit of measurement for fonts), resolution for a specific device and in some cases, a particular face.

As you might imagine, these are very fast. That is, they can be rendered on a device almost instantaneously. This is because they are of a fixed size and already have the pel on/off mapping done. All that needs to be done is to copy the bits to the screen to make them visible. In the case of a printer, the bitmap is sent to the printer if the font is not resident.

The drawback is that these bitmapped fonts are inflexible. Because they are fixed-size maps, they cannot be rotated or scaled. This restricts the user to being able to use only those faces and point sizes that are physically in the font file, or physically resident in the output device (such as a printer).

Vector Font Technology

This is a term used to describe the font technology in OS/2 versions 1.1 and 1.2. Each character is drawn as a series of lines, literally. The hollow character then is filled graphically. A character is not stored as a bitmap, but as a series of vectors. When the character is to be rendered, the lines are drawn on the screen to show the character, then the area is filled.

The advantage to this method is that it is very flexible. Since the character is simply a series of lines and algorithms, it can be rotated and scaled very easily. The drawbacks of this technology lie in performance and quality. Because the lines have to be scaled and drawn in real time on the screen, these fonts are very slow. Once the lines have been drawn, the outline then must be filled. Additionally, there is no way to store a character once it has been drawn.

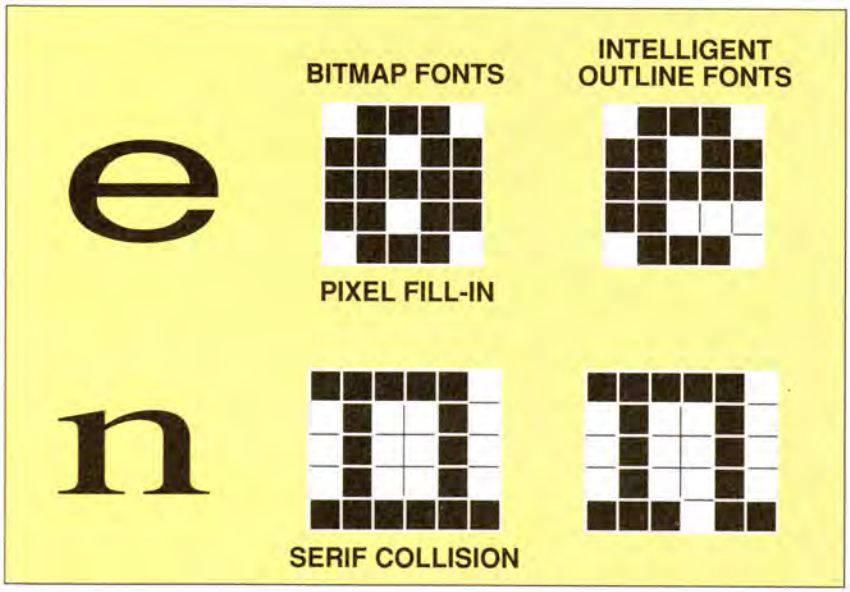


Figure 1. Bitmap vs. Outline Fonts

Adobe Type Manager

In the Summer 1991 issue of the *Developer*, the spotlight article on Adobe® tells how ATM came into OS/2. IBM evaluated the ATM and decided to work with Adobe to include their technology into OS/2. Please refer to that article for details on the makings of ATM. This article concentrates on writing programs to use fonts.

To complete the definitions, these types of fonts are called Type 1 fonts or outline fonts (and sometimes are referred to as Adobe fonts). Note that these are not formal definitions, but ones that have been adopted to differentiate among the various technologies.

In short, ATM is a technology that has been provided with OS/2 that allows fast, flexible, high quality fonts to be used. These fonts use the same basic concept as the vector fonts described above, but with a few advantages over vector fonts.

The Adobe Type Manager is a cross between vector and bitmap font technologies. Adobe fonts use a method of "hints" and algorithms to generate the outlines. These outlines then are filled in much the same way as with vector fonts. The difference is that this is all done in memory, then the character is rasterized to create a bitmap. It is this bitmap that is displayed on the output device.

This technique provides fast, high quality, scalable fonts at all point sizes. The algorithms used are high performance, and have been improved further since their introduction, allowing the fonts to be rendered quickly. (Actually, it is the ATM that has undergone the enhancements. The Type 1 font format has remained constant. OS/2's Intelligent Font Interface allows different types of font engines; the one provided with OS/2 is the ATM engine.)

Another addition to the ATM mechanism of OS/2 is a font cache. Each time a character is rendered in a certain typeface and point size, the character is stored in a cache. If the character in that face and at that size is needed again, and it has not been flushed from the cache, it is retrieved from there, further increasing the speed. This can be done because once the character has been rendered, it is simply a bitmap. The cache is part of the graphics engine. It is 512 Kb and can store up to 40 fonts of 128 characters each.

FONT ARCHITECTURE

A brief description of the architecture of the font subsystem is given here to aid in understanding why we have made certain recommendations in the programming of fonts.

Device and Non-device Fonts

Device fonts are those which physically reside in the hardware of the output device. In the case of the printer, these can be fonts in the printer's system board, or on a font cartridge or card that can be inserted into the printer. Note that device fonts are not equivalent to the downloadable fonts mentioned earlier. Downloadable fonts are non-device fonts.

Non-device fonts are those that do not reside in the output device. For example, bitmapped fonts or fonts that have been installed for the screen but not the printer (and don't reside in the printer's hardware) are non-device fonts.

When referring to device and non-device fonts, you also must be specific regarding which device you are referring to. For example, a font may reside in a printer but not on the display hardware. As such, when you are dealing with a printer device context, the font may be a device font, but when you are drawing with this font on the screen, you are dealing with a non-device font.

Device fonts are inherently faster than non-device fonts because they reside in the hardware of the device. OS/2 and the presentation drivers need to do no work to use them other than to tell the device to set the font.

Looking at Figure 2, you can see how device as well as non-device fonts are rendered. The process begins with an application's requesting that text be drawn on a device. Inside the graphics engine, the device is queried to see if the desired font is a device or non-device font. This query is made of the presentation driver for the device.

If the font requested is a device font (as returned by the presentation driver), it is a simple matter of the engine telling the presentation driver to tell the device to set the font active. If the font requested is a non-device font, some more work must be done at the engine layer to determine the best way to render the character.

Without going into the low-level details here, the presentation driver is asked to set a font. It is up to the driver to either set the font if it is a device font, download the font if it is downloadable, or ask the engine for help in simulating the character with a bitmap. In this case, the font is simulated based on other known quantities about the font. This information is held in a font metrics structure.

Every font has a metrics structure associated with it that describes the font. If an exact match is not found, the engine uses this information to do its best to make the font look as close as possible to the one requested. A bitmap is generated and sent to the device in this instance.

This takes a bit longer than device fonts as there is more work that must be done beforehand, in addition to more data being sent to the printer. A bitmap is larger than a command to set a font. In order of speed, a device font is fastest, a downloadable font is second, and the simulated bitmap font is the slowest of the three.

Although the method for getting the fonts onto the printer may vary from vendor to vendor, OS/2 follows the same progression in determining how to render a character on a device. For example, some vendors may not wish to download fonts and just let the engine simulate bitmaps for them. This is more prevalent in dot-matrix printers. Laser printers, on the other hand, most likely will use downloadable fonts, or will have a font cartridge or card in the hardware. These types of printers are of higher-quality than dot-matrix printers, so simulated bitmaps are less desirable than fonts designed for the printer.

CHOOSING FONTS IN PROGRAMS

Beginning with OS/2 2.0, there is a standard font selection dialog that will query fonts and present them in a list from which the user may choose. This will simplify the font selection function in programs significantly from what was available in 16-bit OS/2. Even with this new standard dialog coming in version 2.0, the discussion presented here will help you to understand the overall font architecture in both the 16- and 32-bit versions of OS/2.

The progression of functions for choosing fonts should be as follows:

- Open a Device Context (DC) based on the current printer selected. At this point it is not necessary to determine all of the printer properties. Only those that are needed to determine character size, such as resolution and paper orientation, are important.
- Next, create a Presentation Space (PS) associated with the DC just created.
- The standard font dialog then is invoked to display the available fonts for the PS and DC.

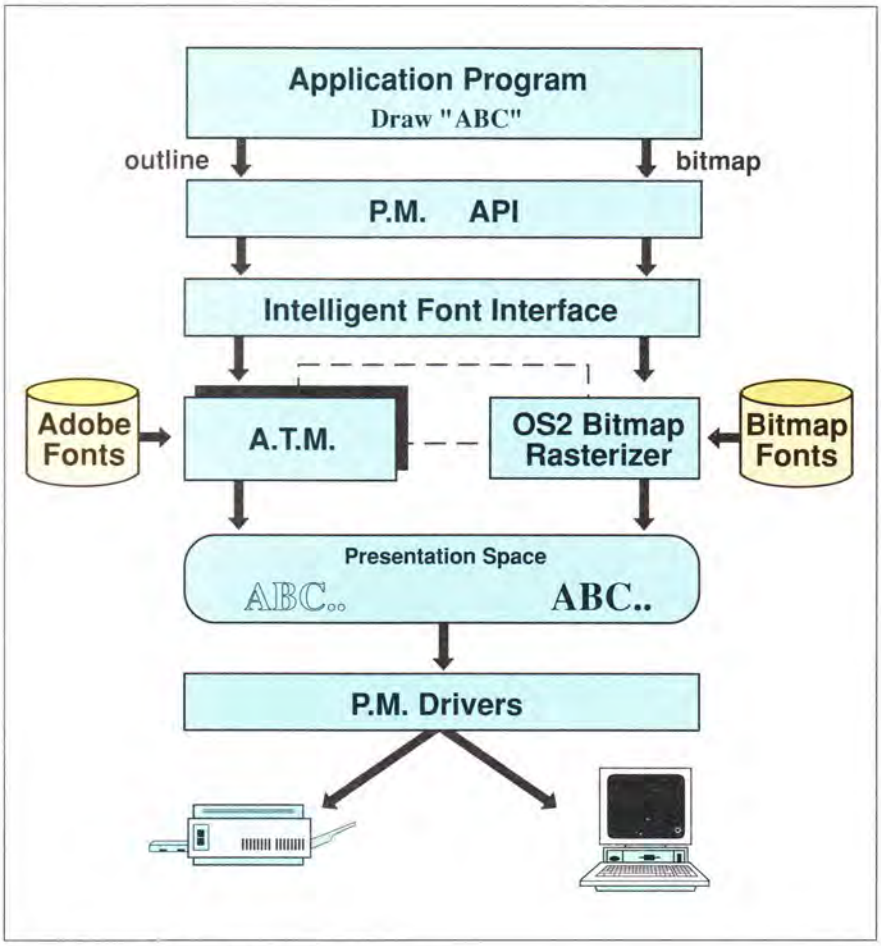


Figure 2. Basic Font Rendering Architecture

- Once the font is selected, you may proceed to work with the DC and PS as desired. You may wish to create generic ones for font determination and then destroy them, or you may decide to create and use them directly.

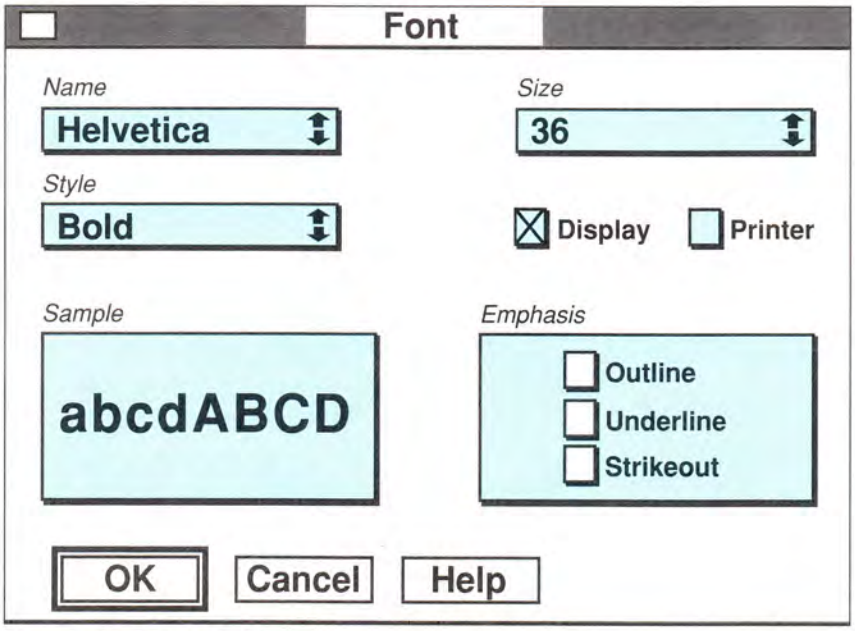


Figure 3. OS/2 2.0 Standard Font Dialog

Note here that the primary difference between OS/2 2.0 and 16-bit OS/2 is the third step listed above. In the 16-bit world, the font dialog is not available, so it is up to the application to query the fonts and display the list in a coherent form to the user. Figure 3 shows what a standard font dialog looks like.

Calling up the Font Dialog

For simplicity, this article will refer to the companion discussion in this issue, "Programming Printing in OS/2", for functions not repeated here.

The first order of business is to create a device context and a presentation space as stated previously. Using the code in the companion article, you can start by having the user select a queue. From that, a device context and a presentation space need to be created. One suggestion is to give the user a choice of queues when the application is first started. You might put this in a "New Document" type of menu option when a new document is created.

```
HDC         hdcPrn;                 /* Printer DC handle */
FONTDLG     fntdFontDStruct;        /* FONTDLG structure for WinFoltDlg call */
DEVOPENSTRUC dopPrn;                /* Structure for DevOpenDC for printer */
BOOL        Rc = TRUE;
HPS         hps;                    /* PS handle for printer */
HDC         hDC;
LONG        alDevinfo[2];
LONG        lDevX, lDevY;

memset((PVOID)&dopPrn,'\0',sizeof(DEVOPENSTRUC)); /* Clear the DEVOPENSTRUC */
memset((PSZ)&fntdFontDStruct,'\0',sizeof(FONTDLG));/* clear font dlg struct */

fntdFontDStruct.hpsScreen = WinGetPS(hWnd);/* PS to allow WinFontDlg to draw */
fntdFontDStruct.cbSize = sizeof(FONTDLG);  /* size of the structure */
fntdFontDStruct.pszPreview = "Hi There";   /* Font sample string */
fntdFontDStruct.pszTitle = "Font Selector"; /* Title of selection window */
fntdFontDStruct.pszPtSizeList = "6 8 10 12 14 16 18 20 24 32 40 48";
fntdFontDStruct.fl = FNTS_CENTER | FNTS_HELPBUTTON; /* FNTS_* flags */
fntdFontDStruct.clrFore = CLR_NEUTRAL;    /* Foreground color for sample */
fntdFontDStruct.clrBack = CLR_BACKGROUND; /* Background color for sample*/
fntdFontDStruct.usWeight = FWEIGHT_NORMAL; /* Weight and width for sample */
fntdFontDStruct.usWidth = FWIDTH_NORMAL;
fntdFontDStruct.fxPointSize  = MAKEFIXED(12, 0);
```

Figure 4. Creating a DC, PS and Calling WinFontDlg (Continued)


```
 /*************************************************************/
 /* Now we need to initialize the DEVOPENSTRUC to create the DC */
 /* These values are obtained from the print destination       */
 /* selection described in the companion article               */
 /*************************************************************/
dopPrn.pszLogAddress = szQueueName;
dopPrn.pszDriverName = szDriverName;
dopPrn.pdriv = (PDRIVDATA)DriverData;
dopPrn.pszDataType = "PM_Q_STD";

   /*******************************************************/
   /* Now open device context for printer                 */
   /*******************************************************/
   hdcPrn = DevOpenDC(hAB,OD_INFO,(PSZ)"*",4L,
                      (PDEVOPENDATA)&dopPrn,
                      (HDC)NULL);

   /*************************************************************/
   /* The 0's here indicate the default size for a PS           */
   /*************************************************************/
     sizlPS.cx = 0L;
     sizlPS.cy = 0L;

     DevQueryCaps( hdcPrn, CAPS_HORIZONTAL_FONT_RES, (LONG)2, alDevinfo);
     lDevX = (LONG)alDevinfo[0];
     lDevY = (LONG)alDevinfo[1];


   /* Create PS associated with the device context */
   fntdFontDStruct.hpsPrinter = GpiCreatePS(Parm->hAB,
                                hdcPrn,
                                &sizlPS,
                                PU_PELS | GPIA_ASSOC | GPIT_MICRO);


if (!WinFontDlg(HWND_DESKTOP, hWnd, &fntdFontDStruct))
   Rc = FALSE;

if (fntdFontDStruct.lReturn == DID_CANCEL)
   Rc = FALSE;

WinReleasePS(fntdFontDStruct.hpsScreen);
```

Figure 4. Creating a DC, PS and Calling WinFontDlg

Once the queue is selected and the PS and DC are created, a font dialog structure must be set up for the call to WinFontDlg. This is a FONTDLG structure as defined in the OS/2 header file, PMSTDDLG.H. In order to have the standard dialog display the list of available fonts, only some of the fields in the FONTDLG structure need to be filled in. An example of creating a Presentation Space and a Device Context, and then calling up the font dialog is shown in Figure 4. Note that the assumption is made that a print destination has been obtained as previously described.

The WinFontDlg Function

The code shown in Figure 4 is used to allow the user to select a font. The first part of the figure shows the local variables needed. The two most important items are the FONTDLG structure and the DEVOPENSTRUC.

Both of these items are defined in the OS/2 header files. The FONTDLG structure contains all of the fields necessary to allow the WinFontDlg function to display the font selection dialog and return the selected information to the program. The DEVOPENSTRUC is a structure used for opening device contexts, in this case, one for the printer.

The first part of the code clears out the entire FONTDLG and DEVOPENSTRUC structures. This must be done because only some of the fields will be filled in prior to the structure's use. The other fields must be blank so the functions do not assume that garbage left in that memory is data.

```
/*********************************************************/
/* When the new font is selected.....                    */
/*********************************************************/


/*********************************************************/
/* Note that pMetrics is what was returned in the        */
/* pMetrics element of the FONTDLG structure returned    */
/* from the call to WinFontDlg.                          */
/*********************************************************/


SHORT iIndex;
PMYSTATUS    pMyStatus;
PFONTMETRICS pMetrics;
CHAR  szTempSize[10];

szTempSize[0] = 0;
pMetrics = (PFONTMETRICS)NULL;

   /*****************************************************/
   /* Load up the string to be displayed                */
   /*****************************************************/

  strcpy(szText,"Sample Text");
  uslen=strlen(szText);

  WinSetDlgItemText(hwnd,IDD_FNT_REC,szText);

    USHORT usCodePage = 0;
    USHORT usDataLen = 0;


   /*****************************************************/
   /* Get the proper codepage to ensure text is         */
   /* Displayed with correct characters for different   */
   /* countries                                         */
   /* GpiQueryCp or WinQueryCp could also be used here  */
   /*****************************************************/
```

Figure 5. Using The Selected Font (Continued)


```
    DosGetCp(2, &usCodePage, &usDataLen);
    if (!usDataLen) usCodePage = pMetrics->usCodePage;

    fattrs.usRecordLength = sizeof(FATTRS);
    fattrs.fsSelection = 0;
    fattrs.lMatch = 0L;
    fattrs.idRegistry = pMetrics->idRegistry;
    fattrs.usCodePage = usCodePage;


/*********************************************************/
/* If we are dealing with an outline font                */
/* Set up the fattr structure as such                    */
/*********************************************************/
    if (pMetrics->fsDefn & FM_DEFN_OUTLINE)
    {
      USHORT usPointSize;
      PBYTE  pb;
      extern LONG lScreenXResolution;
      extern LONG lScreenYResolution;

      usPointSize=atoi(szTempSize);
      if (usPointSize == 0) usPointSize = 12;

      fattrs.lMaxBaselineExt = (((LONG)(usPointSize*10)) *
                                lScreenYResolution) / 720L;
      fattrs.lAveCharWidth =  (((LONG)(usPointSize*10)) *
                               lScreenXResolution) / 720L;

/*********************************************************/
/* The above 2 are needed for MLE's only. This is not    */
/* ordinarily needed.                                    */
/*********************************************************/

      fattrs.fsFontUse = (FATTR_FONTUSE_NOMIX |
                          FATTR_FONTUSE_OUTLINE);
    }
    else
    /*****************************************************/
    /* We've got a bitmap font                           */
    /*****************************************************/
    {
      fattrs.lMaxBaselineExt = pMetrics->lMaxBaselineExt;
      fattrs.lAveCharWidth = pMetrics->lAveCharWidth;
      fattrs.fsFontUse = FATTR_FONTUSE_NOMIX;
    }
    fattrs.fsType = 0;
```

Figure 5. Using The Selected Font (Continued)


```
/*********************************************************/
/* In this case, we are dealing with an MLE, so just     */
/* send the MLM_SETFONT message. For text drawing we      */
/* would use GpiSetCharBox followed by GpiSetLogFont and  */
/* GpiCharStringAt                                        */
/*********************************************************/

  WinSendDlgItemMsg(hwnd,IDD_FNT_SAMPLE,MLM_SETFONT,
                    (MPARAM)&fattrs,(MPARAM)NULL))
```

Figure 5. Using The Selected Font

The next task is to fill in the structures. You can see that the information being supplied by the program is very obvious. There is a list of point sizes to be made available, some sample text to show the user what the font will look like, the color in which to display the font, and a handle to a presentation space among other things.

The information for the DC is the same as that obtained when the print destination is selected. Again, please refer to the companion article in this issue for that code. Once the DC is opened, a PS needs to be created and associated with it. This is accomplished with a call to GpiCreatePS.

Now that there is a DC with a PS for the selected printer, the next step is to call WinFontDlg. Passed to this function is the FONTDLG structure created previously, which contains, among the other items, the handle to the presentation space for the printer. This is how WinFontDlg can determine the fonts available for the device.

In 16-bit OS/2, the steps are the same up until the call to WinFontDlg. The functions there will have to be written by the application programmer. This font dialog is one of several new standard dialogs implemented in OS/2 2.0. The dialog will return a value similar to other dialogs in the Presentation Manager. If DID_CANCEL is returned, the user has pushed the "Cancel" button on the dialog. If the return code from WinFontDlg is other than 0, some error has occurred. If neither of these statements are true, the FONTDLG structure is completely filled in with the information about the font selected.

Once the font has been selected, the application must set up the font to be used. The code in Figure 5 is an example of how to use the font. Note that this particular piece of code simply will change the font in a multi-line edit field. This code easily can be adapted to do general drawing in any window by using the GpiCreateLogFont and GpiSetCharBox functions followed by the GpiCharStringXX functions.

RENDERING CHARACTERS USING A NEW FONT

As you can see in Figure 5, this code is used to display the sample text when the user selects a new font. In this instance, a multi-line entry field (MLE) is used to display the text. Once the face and size are known (which are returned from the call to WinFontDlg), the metrics are retrieved from the structure provided.

The next step is to retrieve the codepage for the system. This is important to set in the FATTRS structure, so that the proper characters for the output device, and especially the country, are displayed.

Now, all of the information for the FATTRS structure is available, so the creation of the structure can take place. The first step is to test for an outline font, and if one is being used, to set up the structure accordingly.

For outline fonts, the fsFontUse portion of the structure must have FATTR_FONTUSE_OUTLINE specified. This indicates to the engine that the outline fonts are requested. This also will enable the new function of the 4019 and Laserjet printer drivers mentioned earlier.

If a bitmap font is being used, FATTR_FONTUSE_OUTLINE is not specified. The average character width and baseline extents are part of the bitmap font metrics structure. The FATTR structure can be filled with these items directly.

The final step is to display the text. This is done in this example with the MLM_SETFONT message. Because this is a small, simple text display, an MLE is easier to work with. A static text field would have sufficed as well.

In many cases, you will need to draw text in plain windows, so system defined messages such as MLM_SETFONT will not help. In these cases, you will need to use the structures and the same basic methods presented here, but instead of sending an MLM_SETFONT message, you will call GpiCreateLogFont, and then call the APIs to draw the text, formatting with line and word breaks as necessary.

As a matter of fact, that is what the MLE code does behind the scenes. When the MLM_SETFONT message is received, the font specified is queried, then the Gpi calls are made to set the logical font and draw the text. This is part of the default behavior of multi-line entry fields.

SENDING PRINT JOBS WITH FONTS

There is no difference between drawing text to the screen and the printer. Once the fonts have been queried, presented to the user and set, the drawing becomes a trivial task.

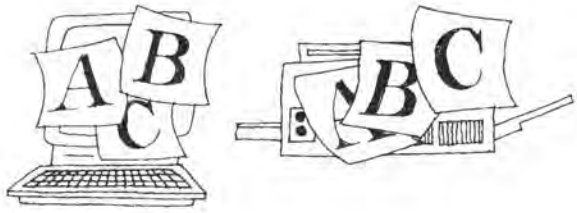
Throughout this article, the assumption is made that the fonts being queried are for a printer. The device fonts being spoken of refer to printer device fonts. There is a reason for this.

Recalling the opening statements in this article, most users want the highest quality output on paper. The way to do this is to make the most use of what the printer has to offer. By querying the fonts available for the printer, you are assured of getting the best possible print quality and speed. Don't disable the use of generic fonts, however, as you may be limiting the overall function in trying to make everything perfect. You will want to retain flexibility in the case of a Type 1 font not being present.

You may ask about the quality of the display. This usually is of less concern for several reasons. First, most displays' resolutions are not as high as a printer's, so the resolution on the screen will be lower in any case. Also, if the font has to be simulated by a bitmap on the screen, it is faster than having to send the bitmap to the printer.

Another item to note is to not use the lMatch value directly in specifying fonts to be printed. The reason is that when you ask for a font with a specific lMatch and there is no exact match, the engine will be asked to simulate one for you. By using the face names and point sizes, you retain more flexibility in asking for fonts on different devices, such as taking what is on the screen and sending it to a printer.

Back to sending the print job. Because a DC for the printer has been created, and a PS created and associated with the DC, your application needs only to call GpiSetLogFont and draw the text into the PS. The translation to the device takes place here along with the DC.



Whether the PS is for a screen DC or a printer DC is of no concern. That is to say, the mechanism for drawing into a PS is the same regardless of the device. The same positioning and other calculations based on the device's page size apply equally. The only thing the application really needs to know at that point is a page size, so it knows when to send a new page command. Please refer to the companion article on printing in this issue for details.

SUMMARY

Fonts are one very apparent way that OS/2 provides device independence for applications. Programs only need to know a few details about the output device, such as screen or printer resolution (for selecting appropriate bitmap fonts) and page size. All of the other work is done by the presentation drivers.

An application does not need to know if a font selected is a device font, a downloadable font, or bitmap font in order to render it on a device. All the application needs to know is that the font is available. Granted, some applications may desire to know the font structure information in order to provide users with higher quality and speed, but the downloading of the font, or the sending of the bitmap font, or the setting of the device font is something that the presentation driver manages. Applications just need to say, "Hey, you've got this font available. Give it to me."

With the font selection dialog coming in version 2.0, the job of presenting the user with a selection list is much simpler than in 16-bit OS/2, but understanding what goes on inside is still vital. Combining this information with that in the companion articles in this issue and the previous one, you will have a comprehensive understanding of the overall OS/2 Print Subsystem.

REFERENCES

IBM OS/2 Presentation Manager Programming Reference, Vol. 1. (S10G-2625).

Bernath, David A. **File and Font Dialogs: Standardized Selection Techniques**, *IBM Personal Systems Developer*, page 18, Winter 1992.

Reich, David E. **Printing Using OS/2**, *IBM Personal Systems Developer*, page 101, Fall 1991.

Reich, David E. **The OS/2 Print Subsystem Architecture**, *IBM Pesonal Systems Developer*, page 107, Fall 1991.

Reich, David E. **Programming Printing Under OS/2**, *IBM Personal Systems Developer*, page 85, Winter 1992.

Cheatham,Paul W. , David E. Reich and Robert F.G. Robinson, **OS/2 Presentation Manager Programming**, John Wiley & Sons, Inc. (ISBN 0-471-50897-7)

David E. Reich, *IBM Corporation, Internal Zip 1424, 1000 NW 51st Street, Boca Raton, FL 33429, is a senior associate programmer working with OS/2 Technical Support in Boca Raton. He has an MS in Computer Science from the State University of New York at Albany, has written several articles previously in "IBM Personal Systems Developer", and has co-authored a book entitled,* ***OS/2 Presentation Manager Programming****, published by John Wiley & Sons, Inc.*

Presentation Manager

Object-Oriented Programming in OS/2 2.0

by Roger Sessions and Nurcan Coskun

Object-Oriented Programming (OOP) is quickly establishing itself as an important methodology in developing high quality, reusable code. In the 2.0 release of OS/2, IBM is introducing a new system for developing class libraries and object-oriented programs. This system is called SOM for System Object Model. This article gives a general introduction to the object-oriented paradigm, discusses developing object-oriented class libraries using SOM, and compares SOM libraries to those developed using standard object- oriented languages.

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

The latest revolution to hit the software community is object-oriented programming. Object-Oriented Programming Languages (OOPL) are being used throughout the industry, Object-Oriented Databases (OODB) are starting elicit widespread interest, and Object-Oriented Design and Analysis (OODA) tools are changing the way people design and model systems. Object-oriented programming is best understood in contrast to its close cousin, structured programming. Both attempt to deal with the same basic issue, managing the complexity of ever more complex software systems.

Structured programming models a system as a layered set of functional modules. These modules are built up in a pyramid-like fashion, each layer representing a higher level view of the system. Structured programming models the system's behavior, but gives little guidance to modeling the system's information.

OOP models a system as a set of cooperating objects. Like structured programming, it tries to manage the behavioral complexity of a system. OOP, however, goes beyond structured programming by also managing the informational complexity of a system.

Because object-oriented programming models both the behavioral and informational complexity of a system, the system tends to be much better organized than if it was simply well "structured". Because object-oriented systems are better organized, they are easier to understand, debug, maintain, and evolve. Well-organized systems also lend themselves to code reuse.

OOP sees the dual issues of managing informational and behavioral complexity as closely related. Its basic unit of organization is the object. Objects have some associated data, which we call the object's state, and a set of behaviors, which we call the object's methods. A class is a general description of an object, which defines the data which represents the object's state, and the methods the object supports.

OBJECT-ORIENTED PROGRAMMING IN C

Before we examine SOM, let's consider object-oriented programming in C; this will lead us naturally into the SOM philosophy. The techniques in this section as well as many related advanced C coding techniques are discussed in the book *Reusable Data Structures for C* [Sessions, '89].

Roger Sessions

Nurcan Coskun

Consider a data structure definition containing information related to a generic stack. We may have a series of functions all designed to operate on our stack structure. Given a basic stack definition, we may have multiple instances of this structure declared within our program.

The following is an example of a generic stack definition in C:

```
struct stackType {
  void *array[STACK_SIZE];
  int top;
};
typedef struct stackType Stack;
```

The following is an example of generic stack functions:

```
Stack *create();
void *pop(Stack *stk);
void push(Stack *stk, void *item);
```

Most C programmers can imagine how such functions would be written. The following is an example of the `push()` function:

```
void push(Stack *stk, void *item)
{
  stk->array[stk->top] = item;
  stk->top++;
}
```

A client program might use this stack to create a stack of words needing interpretation:

```
main()
{
  Stack *wordStack;

  char *subject = "Emily";
  char *verb = "eats";
  char *object = "ice cream";
  char *nextWord;

  wordStack = create();
  push(wordStack, object);
  push(wordStack, verb);
  push(wordStack, subject);

  /* ... */
  while (nextWord = pop(wordStack)) {
    printf("%s\n", nextWord);
    /* ... */
  }
}
```

Using this example, let's look at the language of object-oriented programming. A class is a definition of an object. The definition includes the data elements of the object and the methods it supports. A `Stack` is an example of a class. We say that a stack contains two data elements (`array` and `top`), and supports three methods, `create()`, `push()`, and `pop()`. A method is like a function, but is designed to operate on an object of a particular class. An object is a specific instance, or instantiation, of a class. We say `wordStack` is an object of class `Stack`, or `wordStack` is an instance of a stack.

Every method needs to know the specific object on which it is to operate. We call this object the target object, or sometimes the receiving object. Notice that each method (except `create()`) takes as its first parameter a pointer to the target object. This is because a program may have many objects of a given class, and each are potential targets for the class methods.

There are three important advantages of this type of organization. First, we are developing some generic concepts, which can be reused in other situations in which similar concepts are appropriate. Second, we are developing self-contained code, which can be fully tested before it is folded into our program. Third, we are developing encapsulated code, the internal details of which are hidden and of no interest to the client. Our client `main()` program need know nothing about the `Stack` class other than its name, the methods it supports, and their interfaces.

INTRODUCTION TO SOM

OS/2 2.0 includes a language-neutral object-oriented programming mechanism called System Object Model (SOM). Although it is possible to write object-oriented programs in traditional languages, such as we did with the stack example, SOM is specifically designed to support the new paradigm and to be usable with both procedural (or non object-oriented) languages and object-oriented languages.

A major claim of object-oriented programming is code reusability. This is most often achieved through the use of class libraries. Today's library technology is limited in that these class libraries are always language specific. A C++ library cannot be used by a Smalltalk ™ programmer, and vice-versa. Clearly there is a need to create a language-neutral object model, one which can be used to create class libraries usable from any programming language, procedural or object-oriented. SOM is designed to address this need.

SOM introduces three important features lacking in most procedural languages. These are encapsulation, inheritance, and polymorphism (referred to here as "override resolution"). Encapsulation means the ability to hide implementation details from clients. This protects clients from changes in our implementation, and protects our implementation from tinkering by clients. Our stack example was not protected. Although clients did not need to know the internal data structures of the stack, we had no way to prevent clients from looking at such implementation details. We could discourage, but not prevent, clients from writing code which used, and possibly corrupted, internal stack data elements.

Inheritance, or class derivation, is a specific technique for developing new classes from existing classes. It allows one to create new classes which are more specialized versions of existing classes. For example, we could create a `DebuggableStack`, which is like a `Stack` class, but supports further debugging methods, such as `peek()` and `dump()`.

Inheritance also allows code consolidation. If we have a class defining `GraduateStudent` and `UnderGraduateStudent`, we can consolidate common code into a third class, `Student`. We then define `GraduateStudent` and `UnderGraduateStudent` as more specialized classes, both derived from the common parent `Student`.

Inheritance introduces some additional semantics beyond those we have already examined. A specialized class is said to be derived from a more generalized class. The general class is called the parent, or sometimes, the base class. The specialized class is called the child, or sometimes, the derived class. A child class is said to inherit the characteristics of its parent, meaning that any methods defined for a parent are automatically defined for a child. Thus because `GraduateStudent` and `UnderGraduateStudent` are both derived from `Student`, they both automatically acquire any methods declared in their common parent.

Override resolution means invoked methods are resolved based not only on the name of the method, but also on a class's place within a class hierarchy. This allows us to redefine methods as we derive classes. We might define a `printStudentInfo()` method for `Student` and then override, or redefine, the method in both `UnderGraduateStudent`, and `GraduateStudent`. Override resolution means that the method is resolved based on the type of the target object. If the target object type is a `Student`, the `Student` version of `printStudentInfo()` is invoked. If the target object type is a `GraduateStudent`, the `GraduateStudent` version of `printStudentInfo()` is invoked.

We will now look at SOM in more detail by examining how classes are defined in SOM, how SOM methods are written in the C programming language, and how clients use SOM classes. The intent of SOM is to eventually allow developers to write methods in a variety of languages including the popular object-oriented programming languages. In OS/2 2.0, SOM support is limited to C, thus the language used in the examples.

DEFINING CLASSES IN SOM

The process of creating class libraries in SOM is a three step process. The class designer defines the class interface, implements the class methods, and finally loads the resulting object code into a class library. Clients either use these classes directly, make modifications to suit their specific purposes, or add entirely new classes of their own.

In SOM we define a class by creating a class definition file. We will give a basic example here, and defer more detailed discussion of the many keywords and options to the IBM SOM publication.

The class definition file is named with an extension of `csc`. In its most basic form, the class definition file is divided into the following sections:

1. **Include section:** This section declares files which need to be included, much like the C `#include` directive.
2. **Class name and options:** This section defines the name of the class and declares various options.
3. **Parent information:** This defines the parent, or base, for this class. All classes must have a parent. If your class is not derived from any of your own classes, then its parent will be the SOM defined class `SOMObject`, the class information of which is in the file `somobj.sc`.
4. **Data Section:** This section declares any data elements contained by objects of this class. By default, data can be accessed only by methods of the class.

```
include <somobj.sc>

class:
  Student;

- "Student" class provides a base class to generate more
- specialized students like "GraduateStudent" and
- "UnderGraduateStudent".

parent:
  SOMObject;

data:
  char  id[16];       /* student id */
  char  name[32];     /* student name */

methods:

  void  setUpStudent(char *id, char *name);
  - sets up a new student.

  void  printStudentInfo();
  - prints the student information.

  char  *getStudentType();
  - returns the student type.

  char  *getStudentId();
  - returns the student id.
```

Figure 1. Class Definition File: <student.csc>

5. **Methods Section:** This section declares methods which objects of this class support. By default, all methods declared in this section are available to any class client.

Comments can be used for documentation purposes, and the following styles are all acceptable:

```
/* This is a comment. */
// This is a comment.
— This is a comment.
```

The class definition file, `student.csc`, describes a non-derived `Student` class, and is shown in Figure 1.

WRITING METHODS

Class methods are implemented in the class method implementation file. Each method defined in the method section of the class definition file needs to be implemented. They can be implemented in any language that offers SOM support, which for now is only C. The student class method implementation file, `student.c`, is shown in Figure 2.

Notice that the method code looks much like standard C, with a few differences.

First, each method takes, as its first parameter, a pointer (`somSelf`) to the target object. This is very similar to our C stack implementation. This parameter is implicit in the class definition file, but is made explicit in the method implementation.

Second, each method starts with a line setting an internal variable named `somThis`, which is used by macros within the SOM class header file.

Third, names of data elements of the target object are preceded by an underscore character. The underscored name turns into a C language macro defined in the class header

```
#define Student_Class_Source
#include "student.ih"

static void setUpStudent(Student *somSelf, char *id, char *name)
{
    StudentData *somThis = StudentGetData(somSelf);
    strcpy(_id, id);
    strcpy(_name, name);
}
static void printStudentInfo(Student *somSelf)
{
    StudentData *somThis = StudentGetData(somSelf);
    printf("    Id          : %s \n", _id);
    printf("    Name        : %s \n", _name);
    printf("    Type        : %s \n", _getStudentType(somSelf));
}
static char *getStudentType(Student *somSelf)
{
    StudentData *somThis = StudentGetData(somSelf);
    static char *type = "student";
    return (type);
}
static char *getStudentId(Student *somSelf)
{
    StudentData *somThis = StudentGetData(somSelf);
    return (_id);
}
```

Figure 2. Class Method Implementation File:<student.c>


```
#define Student_Class_Source
#include "student.ih"

static void setUpStudent(Student *somSelf, char *id, char *name)
{
    StudentData *somThis = StudentGetData(somSelf);
}
static void printStudentInfo(Student *somSelf)
{
    StudentData *somThis = StudentGetData(somSelf);
}
/* ...and so on for the other methods. */
```

Figure 3. SOM Compiler Generated <student.c>

file, part of the package SOM offers to shield method developers from the details of memory layout.

Fourth, methods are invoked using an underscored syntax. This underscored name turns into a macro invocation which shields programmers from having to understand the details of method resolution.

The first parameter of every method is always a pointer to the target object. This can be seen in the method `printStudentInfo()` which invokes the method `getStudentType()` on its own target object.

The process of creating a class method implementation file can be greatly speeded up by the SOM compiler, which creates a valid C method implementation file lacking only the body of the methods. The body is then filled in by the class implementor. For the student example, the SOM compiler would create a file similar to the one shown in Figure 3.

`student.csc`	This is the class definition file, as described earlier.
`student.sc`	This is a subset of the class definition file. It includes all information from the `.csc` file which is public, including comments on public elements. For the student example, `student.sc` would include everything from `student.csc` except the data section. This file is created by the SOM compiler, and although human readable, should not be edited, as it will be regenerated whenever changes are made to the `.csc` file.
`student.h`	This is a valid C header file which contains macros necessary to invoke public methods and access public data elements of the class. This file will be included in any client of the class. This file is created by the SOM compiler, and is normally only read by programmers who need to know how method resolution is implemented. This file should not be edited.
`student.ih`	Similar to `student.h`, but contains additional information needed for implementing methods. This is the implementor's version of the `.h` file, and must be included in the class methods implementation file. This file is created by the SOM compiler and should not be edited.
`student.c`	Contains the method implementations. This is initially created by the SOM compiler and then updated by the class implementor.

Table 1. Student Class Files


```
include <student.sc>

class:
  GraduateStudent;

parent:
  Student;

data:
  char  thesis[128];     /* thesis title */
  char  degree[16];      /* graduate degree type */

methods:
  override  printStudentInfo;
  override  getStudentType;
  void  setUpGraduateStudent(
        char *id, char *name, char *thesis, char *degree);
```

Figure 4. Class Definition File:<graduate.csc>

MECHANICS OF USING SOM

There is a set of files involved with each class. Here we will look at the most important of these files and discuss their purpose and how they are created. They have different extensions, but all have the same filename as the class definition file, `student` in our example. The SOM compiler generates files from the class definition based on the value of an environment variable, as described in the SOM users guide [SOM]. These files are described in Table 1.

BUILDING SOM CLASSES FROM OTHER CLASSES

There are two ways to use classes as building blocks for other classes. These are derivation (or inheritance) and construction. Let's consider derivation first.

In this example, `GraduateStudent` is derived from `Student`, its base, or parent class. A derived class automatically picks up all characteristics of the base class. A derived class can add new functionality through the definition and implementation of new methods. A derived class can also redefine methods of its base class, a process called overriding. `GraduateStudent` adds `setUpGranduateStudent()` to those methods it inherits from `Student`. It overrides two other inherited methods, `printStudentInfo()` and `getStudentType()`. It inherits without change `setUpStudent()` and `getStudentId()` from the `Student` base class.

The class definition file for `GraduateStudent`, `graduate.csc`, is shown in Figure 4.

```
#define GraduateStudent_Class_Source
#include "graduate.ih"
static void printStudentInfo(GraduateStudent *somSelf)
{
    GraduateStudentData *somThis = GraduateStudentGetData(somSelf);
    parent_printStudentInfo(somSelf);
    printf("    Thesis     : %s \n", _thesis);
    printf("    Degree     : %s \n", _degree);
}
```

Figure 5. Class Method Implementation File:<graduate.c> (Continued)


```
static char *getStudentType(GraduateStudent *somSelf)
{
    static char *type = "Graduate";
    return (type);
}
static void setUpGraduateStudent(GraduateStudent *somSelf,
    char *id, char *name, char *thesis, char *degree)
{
    GraduateStudentData *somThis = GraduateStudentGetData(somSelf);
    _setUpStudent(somSelf,id,name);
    strcpy(_thesis, thesis);
    strcpy(_degree, degree);
}
```

Figure 5. Class Method Implementation File:<graduate.c>

The method implementation file, `graduate.c`, is shown in Figure 5.

Often an overridden method will need to invoke the original method of its parent. For example, the `printStudentInfo()` for `GraduateStudent` first invokes the `Student` version of `printStudentInfo()` before printing out the `GraduateStudent` specific information. The syntax for this is "`parent_MethodName`", as can be seen in the `printStudentInfo()` method.

A given base class can be used for more than one derivation. The class, `UnderGraduate-Student`, is also derived from `Student`. The class definition file, `undgrad.csc`, is shown in Figure 6.

The method implementation file, `undgrad.c`, is shown in Figure 7.

The second technique for building classes is construction. This means that a class uses another class, but not through inheritance. A good example of construction is the class `Course` which includes an array of pointers to `Students`. Each pointer contains the address of a particular student taking the course. We say that `Course` is constructed from `Student`. The class definition file for `Course`, `course.csc`, is shown in Figure 8.

Often classes will want to take special steps to initialize their instance data. An instance of `Course` must at least initialize the `enrollment` data element, to ensure the array

```
include <student.sc>

class:
  UnderGraduateStudent;

parent:
  Student;

data:
  char  date[16];     /* graduation date */

methods:
  override  printStudentInfo;
  override  getStudentType;
  void  setUpUnderGraduateStudent(char *id, char *name, char *date);
```

Figure 6. Class Definition File: <undgrad.csc>


```
#define UnderGraduateStudent_Class_Source
#include "undgrad.ih"

static void printStudentInfo(UnderGraduateStudent *somSelf)
{
    UnderGraduateStudentData *somThis =
       UnderGraduateStudentGetData(somSelf);
    parent_printStudentInfo(somSelf);
    printf("    Grad Date  : %s \n", _date);
}
static char *getStudentType(UnderGraduateStudent *somSelf)
{
    static char *type = "UnderGraduate";
    return (type);
}
static void setUpUnderGraduateStudent(
    UnderGraduateStudent *somSelf,char *id, char *name, char *date)
{
    UnderGraduateStudentData *somThis =
       UnderGraduateStudentGetData(somSelf);
    _setUpStudent(somSelf,id,name);
    strcpy(_date, date);
}
```

Figure 7. Class Method Implementation File:<undgrad.c>

index starts in a valid state. The method `somInit()` is always called when a new object is created. This method is inherited from `SOMObject`, and can be overridden when object initialization is desired.

This example brings up an interesting characteristic of inheritance, the *is-a* relationship between derived and base classes. Any derived class can be considered as a base class. We say that a derived class *is-a* base class. For example, any `GraduateStudent` *is-a* `Student`, and can be used anyplace we are expecting a `Student`. The converse is not true. A base class is not a derived class. A `Student` can not be treated unconditionally as a `GraduateStudent`. Thus elements of the array `studentList` can point to either `Students`, `GraduateStudents` or `UnderGraduateStudents`.

```
include <somobj.sc>

class:
  Course;

- "Course" class describes the interfaces required to setup the
- course information. The students are "Student" class type and
- can be added to or dropped from the courses through the
- "addStudent" and "dropStudent" methods.

parent:
  SOMObject;
```

Figure 8. Class Definition File: <course.csc> (Continued)


```
data:
  char    code[8];            /* course code number */
  char    title[32];          /* course title */
  char    instructor[32];     /* instructor teaching */
  int     credit;             /* number of credits  */
  int     capacity;           /* maximum number of seats */
  Student *studentList[20];   /* enrolled student list */
  int     enrollment;         /* number of enrolled students */

methods:
  override  somInit;

  void  setUpCourse(char *code, char *title,
     char *instructor, int credit, int capacity);
  - sets up a new course.

  int   addStudent(Student *student);
  - enrolls a student to the course.

  void  dropStudent(char *studentId);
  - drops the student from the course.

  void  printCourseInfo();
  - prints course information.
```

Figure 8. Class Definition File: <course.csc>

The method implementation file for `Course`, `course.c`, is shown in Figure 9.

Notice in particular the method `printCourseInfo()`. This method goes through the array `studentList` invoking the method `printStudentInfo()` on each student. This method is defined for `Student`, and then overridden by both `GraduateStudent` and `UnderGraduateStudent`. Since the array element can point to any of these three classes, we can't tell at compile time what the actual type of the target object is, only that the target object is either a `Student` or some type derived from `Student`. Since each of these classes defines a different `printStudentInfo()` method, we don't know which of these methods will be invoked with each pass of the loop. This is all under the control of override resolution.

THE SOM CLIENT

Now let's see how a client might make use of these four classes in a program. As we look at the program example shown in Figure 10, we can discuss how objects are instantiated, or created, in SOM, and how methods are invoked.

A class is instantiated with the method `classNameNew()`, which is automatically defined by SOM for each recognized class. Methods are invoked by clients just as they are inside SOM methods, and very similarly to our earlier C examples. The first parameter is the target object. The remaining parameters are whatever information is needed by the method. The only odd feature is the underscore preceding the method name, which turns what looks like a regular function call into a macro defined in the `.h` file.


```
#define Course_Class_Source
#include <student.h>
#include "course.ih"

static void somInit(Course *somSelf)
{
    CourseData *somThis = CourseGetData(somSelf);
    parent_somInit(somSelf);
    _code[0] = _title[0] = _instructor[0] = '\0';
    _credit = _capacity = _enrollment = 0;
}
static void setUpCourse(Course *somSelf, char *code,
    char *title, char *instructor, int credit, int capacity)
{
    CourseData *somThis = CourseGetData(somSelf);
    strcpy(_code, code);
    strcpy(_title, title);
    strcpy(_instructor, instructor);
    _credit = credit;
    _capacity = capacity;
}
static int addStudent(Course *somSelf, Student *student)
{
    CourseData *somThis = CourseGetData(somSelf);
    if(_enrollment >= _capacity)  return(-1);
    _studentList[_enrollment++] = student;
    return(0);
}
static void dropStudent(Course *somSelf, char *studentId)
{
    int i;
    CourseData *somThis = CourseGetData(somSelf);
    for(i=0; i<_enrollment; i++)
      if(!strcmp(studentId, _getStudentId(_studentList[i]))) {
        _enrollment--;
        for(i; i<_enrollment; i++)
          _studentList[i] = _studentList[i+1];
        return;
      }
}
static void printCourseInfo(Course *somSelf)
{
    int i;
    CourseData *somThis = CourseGetData(somSelf);
    printf("  %s %s \n", _code, _title);
    printf("  Instructor Name : %s \n", _instructor);
    printf("  Credit = %d, Capacity = %d, Enrollment = %d \n\n",
             _credit, _capacity, _enrollment);
    printf("  STUDENT LIST: \n\n");
    for(i=0; i<_enrollment; i++) {
      _printStudentInfo(_studentList[i]);
      printf("\n");
    }
}
```

Figure 9. Class Method Implementation File:<course.c>


```
#include <student.h>
#include <course.h>
#include <graduate.h>
#include <undgrad.h>
main()
{
  Course *course = CourseNew();
  GraduateStudent *jane = GraduateStudentNew();
  UnderGraduateStudent *mark = UnderGraduateStudentNew();
  _setUpCourse(course, "303", "Compilers ",
     "Dr. David Johnson", 3, 15);
  _setUpGraduateStudent(jane,"423538","Jane Brown",
     "Code Optimization","Ph.D.");
  _setUpUnderGraduateStudent(mark,"399542",
     "Mark Smith", "12/17/92");
  _addStudent(course, jane);
  _addStudent(course, mark);
  _printCourseInfo(course);
}
```

Figure 10. SOM Client Code

When run, the client program gives the output shown in Figure 11.

In the client program output we can see the override resolution at work in the different styles of displaying `UnderGraduate-Students` and `GraduateStudents`. A `Course` thinks of itself as containing an array of `Students`, and knows that any `Student` supports a `printStudentInfo()` method. But the `printStudentInfo()` method that an `UnderGraduateStudent` supports a is different than the `printStudentInfo()` method that a `GraduateStudent` supports, and the two methods give different outputs.

```
303 Compilers
Instructor Name : Dr. David Johnson
Credit = 3, Capacity = 15, Enrollment = 2

STUDENT LIST:

  Id          : 423538
  Name        : Jane Brown
  Type        : Graduate
  Thesis      : Code Optimization
  Degree      : Ph.D.

  Id          : 399542
  Name        : Mark Smith
  Type        : UnderGraduate
  Grad Date   : 12/17/92
```

Figure 11. Client Program Output

COMPARISON TO C++

In this section we will compare some SOM features to those of the most widespread object-oriented programming language, C++, developed by Bjarne Stroustrup. Some good introductory books about object-oriented programming in C++ are *Class Construction in C and C++* [Sessions, 91], *The C++ Programming Language* [Stroustrup], and *C++ Primer* [Lippman].

SOM has many similarities to C++. Both support class definitions, inheritance, and overridden methods (called virtual methods in C++). Both support the notion of encapsulation. But whereas C++ is designed to support standalone programming efforts, SOM is primarily focused on the support of commercial quality class libraries. Most of the differences between SOM and C++ hinge on this issue.

C++ class libraries are version dependent, while SOM class libraries are version independent. When a new C++ class library is released, client code has to be fully recompiled, even if the changes are unrelated to public interfaces. This problem is discussed in detail in the book *Class Construction in C and C++* [Sessions, 91]. SOM, unlike C++, directly supports the development of upwardly compatible class libraries.

C++ supports programming in only one language, C++. SOM is designed to support many languages (although in this first release it supports only C). Rather than a language, SOM is really a system for defining, manipulating, and releasing class libraries. SOM is used to define classes and methods, but it is left up to the implementor to choose a language for implementing methods. Most programmers will therefore be able to use SOM quickly without having to learn a new language syntax.

C++ provides minimal support for implementation hiding, or encapsulation. C++ class definitions, which must be released to clients, typically include declarations for the private data and methods. This information is, at best, unnecessarily detracting, and at worst, proprietary. In SOM, the client never has to see such implementation details. The client need see only the `.sc` files, which by definition contain only public information.

C++ has limited means of method resolution. SOM offers several alternatives. Like C++, SOM supports offset method resolution, meaning that each method is represented by a method pointer which is set once and for all at compile time. Unlike C++, SOM also offers facilities for resolving methods at run time. Name Lookup resolution allows a client to ask for a pointer to a method by method name.

Dispatch resolution allows a client to package parameters at run time for dispatching to a method, a technique which allows SOM to be integrated into interpreted languages, such as Smalltalk.

One other interesting difference between SOM and C++ is in their notion of class. In C++, the class declaration is very similar to a structure declaration. It is a compile-time package with no characteristics that have significance at runtime. In SOM, the class of an object is an object in its own right. This object is itself an instantiation of another class, called the metaclass. The class object supports a host of useful methods which have no direct parallels in C++, such as `somGetName()`, `somGetParent()`, and `somFindMethod()`.

SUMMARY

A new object modeling system is introduced in OS/2 2.0. This object model is called The System Object Model, or SOM. SOM is a dynamic object model which can provide useful class information about objects at run time. The goal of SOM is to support the development of class libraries useful by both compiled and interpreted languages.

ACKNOWLEDGEMENTS

SOM is the work of many people. Mike Conner developed the initial idea and implementation, and continues to lead the overall design of SOM. Andy Martin designed the SOM Class Interface Language, and designed and implemented the class Interface compiler. Larry Raper implemented many features of the run time library and ported SOM to OS/2. Larry Loucks provided close technical tracking and was instrumental in focusing the effort. Early SOM users who contributed much to the evolution of SOM include Nurcan Coskun, Hari Madduri, Andy Martin, Roger Sessions, and John Wang. The project is managed by Tony Dvorak.

REFERENCES

Lippman, Stanley B. *C++ Primer, Second Edition*. Addison-Wesley, Reading, MA, 1989.

Sessions, Roger. *Reusable Data Structures for C*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

Sessions, Roger. *Class Construction in C and C++, Object-Oriented Programming Fundamentals*. Prentice-Hall, Englewood Cliffs, NJ, 1991 (in press).

Stroustrup, Bjarne. *The C++ Programming Language, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1991.

OS/2 2.0 Technical Library: System Object Model Guide and Reference. IBM Publication, 1991 (10G6309).

Nurcan Coskun, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Her expertise is in integrated programming environments, code generators, incremental compilers, interpreters, language based editors, symbolic debuggers, application frameworks, and language design. She is now working on object-oriented programming environments and previously worked on the OS/2 Database Manager. Dr. Coskun can be contacted at nurcan@ausvm1.iinus1.ibm.com. Dr. Coskun has a BS in Industrial Engineering from Middle East Technical University, an MS and a Ph.D. in Computer Science from the University of Missouri-Rolla.*

Roger Sessions, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. He is the author of two books, Reusable Data Structures for C, and Class Construction in C and C++, and several articles. He is working on object-oriented programming environments and previously worked with high performance relational databases and object-oriented storage systems. Mr. Sessions can be contacted at sessions@ausvm1.iinus1.ibm.com. Mr. Sessions has a BA in Biology from Bard College and an M.E.S. in Database Systems from the University of Pennsylvania.*

System Application Architecture

First Impressions are Everything: A CUA-Compliant Installation Program

by Michael Heck and James Rudd

First impressions regarding a particular software application usually are formed during the product's installation procedure. Installation programs that are straightforward, easy to use, and provide a modicum of ingenuity can evoke favorable impressions not only about the installation program, but the software application as well. However, if using the installation procedure is comparable to visiting the dentist for a root canal, it is likely a user will carry these initial negative impressions when it comes time to use the application itself.

A favorable impression is more likely to occur if an installation program is designed around two basic usability principles: *ease-of-use* and *consistency*. An easy-to-use application is one in which the interaction between the application and the user is intuitive and straightforward. A consistent application is one that implements common functions in a manner consistent with other applications (e.g., one application's exit procedure is the same as the exit procedure for other applications). Each of these principles is inherent in the OS/2® Presentation Manager® (PM) and the Common User Access™ (CUA™) architectures.

When developing an OS/2 PM application, developers typically rely upon the CUA guidelines for direction. These guidelines can be found in the *Advanced Interface Design Reference*. Although the CUA documentation offers guidelines for developing CUA-compliant application interfaces that are easy to use and consistent, it offers little substantive guidance for developing easy-to-use CUA-compliant installation programs.

This lack of CUA guidance has resulted in many current OS/2 applications foregoing CUA-compliant installation programs primarily because developers are unsure about what constitutes a CUA-compliant installation program. Moreover, developers who have limited access to iterative usability design and testing often are forced to make trade-off decisions that may or may not improve usability. The focus of this article is to provide the design and code examples of a production-level installation program which has received CUA conformance approval and passed usability testing. It is not intended to be a definitive source upon which all future CUA-compliant installation programs should be measured, but rather an example of how the CUA guidelines can be interpreted and implemented for a given installation program.

Michael Heck

James Rudd

THE DEVELOPMENT PROCESS

The installation program documented in this article was developed for a banking application being migrated from DOS to OS/2. Customer feedback regarding the DOS-based installation program identified problem areas that included poor error recovery, no on-line help support, and the requirement that the user edit the system's CONFIG.SYS and AUTOEXEC.BAT files after the program finished copying files to the hard drive. In essence, like most installation programs for DOS and OS/2, the installation program was little more than a glorified "copy a:*.* c:" statement.

For the OS/2 release, the installation program's end user interface (EUI) was completely redesigned so that it was CUA compliant, took full advantage of the

capabilities of OS/2's Presentation Manager, and above all, was easy-to-use. Based on a structured survey of customers as well as feedback from usability tests conducted on the DOS version, the following features were included in the OS/2 installation program described in this paper:

- Detect previous version of product and alert user.
- Configure user's system automatically, by updating CONFIG.SYS file and creating a program group and title.
- Provide default settings (drive/directory names) for standard installations and allow these settings to be modified.
- Link into the Information Presentation Facility (IPF) to provide general, as well as contextual (context sensitive) help.
- Provide adequate feedback concerning the file transfer status of a component.
- Suppress the standard PM error screens and replace them with more informative message boxes.
- Interface with PKWARE, Inc.'s PKZIP2™ file compression utility for OS/2.

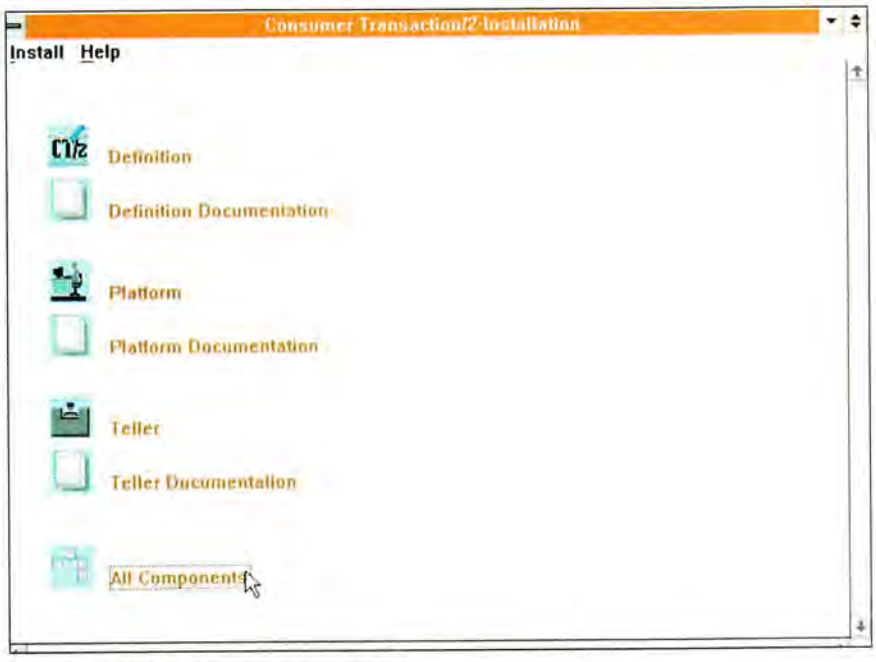


Figure 1. Installation Program Primary Window

- Provide extensive error-checking routines and allow for convenient error-recovery.

In addition to these features, prototypes of the installation program were developed and tested iteratively. This development approach not only enabled us to continually improve the code, but it also provided an opportunity to gain a better perspective on the needs and concerns of our users. The remainder of this article will highlight the major components of this program. Emphasis will be placed on the interaction between the user and the installation program. The features listed above will be discussed in more detail, and we will provide the code listings to some of the more unique features.

INTERACTING WITH THE APPLICATION

After activating the installation program and dismissing the customary "About" dialog box, the user is presented with the primary window depicted in Figure 1. This window is comprised of seven icon/text windows, each of which (except the "All Components" icon/text window) represents a separate component of the product. Each of the components can be selected separately by using the mouse (single or double click) or keyboard (space bar or enter key) as the selection device. The "All Components" option was provided so that users could select all six options with one, rather than six, individual mouse or keyboard actions. Once selected, the component's text window is highlighted (reverse video) to indicate it would be included in the installation process. The selection process in this interface operates much like a "toggle" switch in that deselecting a component requires the same mouse or keyboard action. This causes the highlighted component to return to its unhighlighted state.

Figure 2 shows the primary window of the installation program after the components have been selected and the "Install" menu bar has been activated. As can be seen from this menu bar, the user is presented with five separate options. Selection of the "Display settings..." option produces a dialog box which contains the current settings for each of the selected components. If the user selects the

"Change settings..." option, he or she is presented with a dialog box which contains all of the relevant configuration and system information necessary to install each of the selected components. The "Reset settings..." option allows the user to change the settings back to their default values in the event these settings were changed, whereas the "Start" and "Exit" options allow the user to begin or end the installation program. Customers who were brought in to review the installation prototype indicated to us that having the capability to control system settings explicitly was an important installation program attribute.

Figure 3 depicts the dialog box that is displayed when the user invokes the "Change settings..." option. As you can see, this dialog box provides default settings for each of the selected components (e.g., paths, drives, subdirectories, etc.). Those components not selected for installation are grayed-out and unavailable. From this dialog box the user can customize any of the installation parameters shown. The program also performs error checking and alerts the user when invalid entries are made.

Providing the User With Help

As an additional feature, this installation program offers access to the Information Presentation Facility (IPF) in order to provide both general and contextual (context sensitive) help. One problem area encountered while developing the help structure involved preloading the ".hlp" file. Currently, PM does not provide a method to bind this file with an ".exe" file. Instead, a "help instance" is created and associated when the application is activated, and the ".hlp" file is accessed when help is requested. This arrangement is acceptable when the help file is located on a hard drive, but becomes problematic if the help file is located on multiple diskettes (as is the case with installation programs for most large applications). In this situation, PM only recognizes the help file on the diskette where the "help instance" was created and associated. An error screen is displayed if help is requested on any other diskette, even if that diskette contains the same ".hlp" file. The partial code listing in Figure 4 shows how to avoid this problem by destroying the old "help instance", and creating and associating a new one each time the installation program requests the user to insert a new diskette.

Error Recovery

In interacting with software, users often make mistakes ranging from specifying incorrect system information to simply

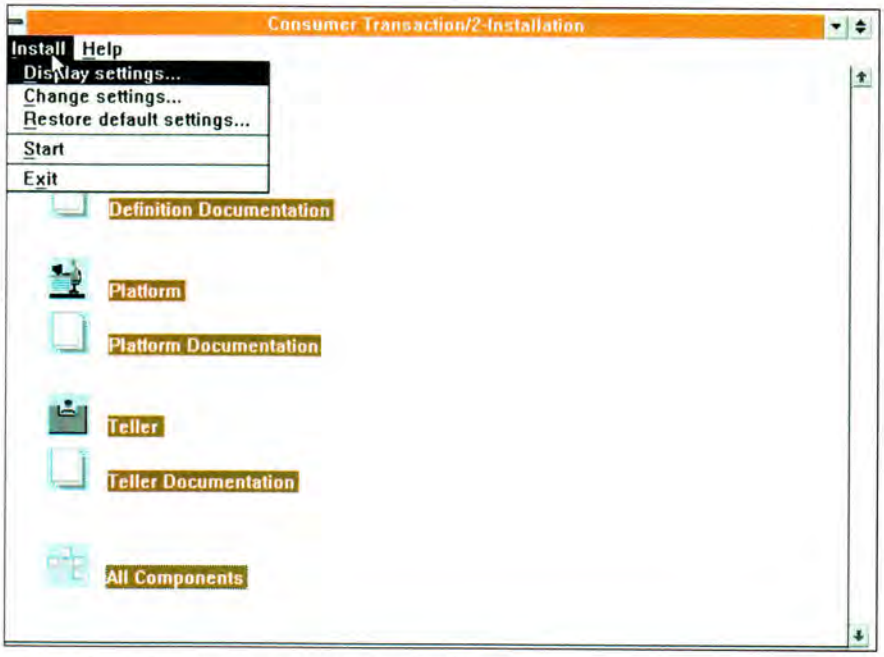


Figure 2. Screen Showing Installation Ready to Install

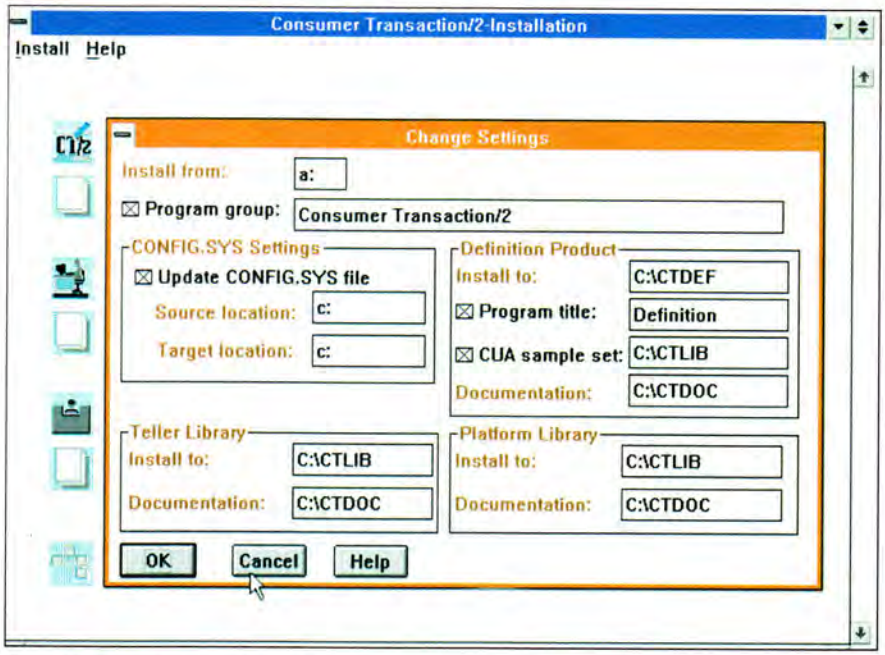


Figure 3. Change Settings Dialog Box

Customers who were brought in to review the prototype indicated to us that having the capability to control system settings was an important installation attribute

```
BOOL cwCopyFileProcedure(CallhWnd, hWndParent)

HWND CallhWnd;
HWND hWndParent;

***** Non-relevant code removed *****

/*********************************************************************/
/* Need to create and associate a help instance for each diskette */
/*********************************************************************/
   if(hWndDZZHFDMWHelp)
   {
     rc = WinDestroyHelpInstance(hWndDZZHFDMWHelp);

     hWndDZZHFDMWHelp = WinCreateHelpInstance(hAB, &hiDZZHFDMWHelp);

     if(hWndDZZHFDMWHelp)
     {
       WinAssociateHelpInstance(hWndDZZHFDMWHelp, hWndFrame);
     }
   }

***** Non-relevant code removed *****
```

Figure 4. Code for Handling Help Instances

forgetting to insert a diskette into the disk drive. Developers are given two choices for handling these mistakes — they can either let PM provide an error message to the user (e.g., full screen trap messages) or they can provide their own error messages. This installation program provides its own customized error messages. Figure 5 represents the code listing used to inform the user when the source disk drive is not ready. After obtaining the correct source drive, the program uses a "DosError-(HARDERROR_DISABLE)" API call to deactivate PM's internal error checking routine. As section (A) of this code listing demonstrates, if an internal error occurs (e.g., the disk drive is not ready), the program displays a message box asking the user to check the disk drive and ensure that a diskette

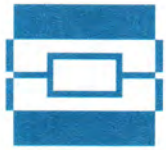
```
USHORT cwFlagCallProcedure(hWndSecondary, hWndPrimary)

HWND hWndSecondary;     // handle of window or d/box.
HWND hWndPrimary;       // handle of Parent

{

 USHORT rc, rcMessage, index, low4, usAction, length, cbRead, rcContinue;
 static HFILE hfCT2INFO;
 static CHAR ct2ArrayTemp[MAXROWS] [MAXCOLUMNS];
 USHORT rc_MBPROGRAMDISK = MBID_OK;
 CHAR *ct2Tmp;
```

Figure 5. Code for Handling System Error (Continued)



```
/*********************************************************************/
/* Begin this function by first changing to the source drive (e.g.,  */
/* a: or b:) and then changing to the 'root' directory.  While this  */
/* is occuring the program will disable OS/2's error handle procedure */
/* (e.g., ugly black text screen that appears whenever a drive       */
/* is not ready, etc., and replace it with our own message structure. */
/* PM's error handling will be reactivated only when the program has  */
/* detected a disk has been correctly inserted into the source drive. */
/*********************************************************************/

     /*******************************/
     /* Obtain current source drive */
     /*******************************/
     low4= (USHORT)(0xFF & tolower(vszSDOInfo[0][0])) - 0x60;

 /*****************************************************************/
 /* Deactivate PM's hard code error routine on a temporary basis  */
 /*****************************************************************/

     rc = DosError(HARDERROR_DISABLE);

     do
     {
      /*********************************************/
      /* Change to source drive and root directory */
      /*********************************************/

      rc = DosSelectDisk(low4);
      strcpy(vszDirData,"\\");
      rc = DosChDir(vszDirData, 0L);

      if (rc != 0)
      {                                                    (A)
        strcpy(vszTemp6,"The ");
        strcat(vszTemp6,vszSDOInfo[0]);
        strcat(vszTemp6," disk drive is not ready.  Make");
        strcat(vszTemp6," sure the disk is inserted correctly");
        strcat(vszTemp6," and is engaged.  Press CANCEL to stop");
        strcat(vszTemp6," the installation.");
        rcMessage = WinMessageBox(HWND_DESKTOP,
                       hWndSecondary,
                       vszTemp6,
                       "Consumer Transaction/2-Installation",
                       0,
                       MB_OKCANCEL | MB_ICONHAND | MB_MOVEABLE);

          if (rcMessage == MBID_CANCEL)
          {
            return(FALSE);
          }

      }
```

Figure 5. Code for Handling System Error (Continued)

```
     } while (rc != 0);

***** Non-relevant code removed *****

     /*******************************/
     /* Reactivate system's DosError */
     /*******************************/

     rc = DosError(HARDERROR_ENABLE);


***** Non-relevant code removed *****
```

Figure 5. Code for Handling System Error

Users resent, for instance, having to look at the disk drive light to discern whether an installation application is still in the process of copying files

is inserted. The "do...while" loop that contains this error routine continually checks the status of the disk drive and does not end until the drive is properly engaged or the user cancels the install routine. Once the condition has been successfully satisfied, PM's internal error checking routine is reactivated.

File Transfer

Once the installation process begins (selecting the "Start" option from the "Install" pulldown), only those components that have been selected and are highlighted will be installed. For each of the selected components, the program checks to see if the component already exists. If a previous version of the component is detected, the installation program alerts the user and asks for confirmation to continue. If the component does not exist, or the user requests a re-installation, the installation program determines if enough disk space is available. In situations where there is not enough disk space, the user is alerted and given the opportunity to continue installing the other components (assuming there's enough disk space) or returning to the primary window. When enough disk space is available, the program asks the user to insert the appropriate diskette.

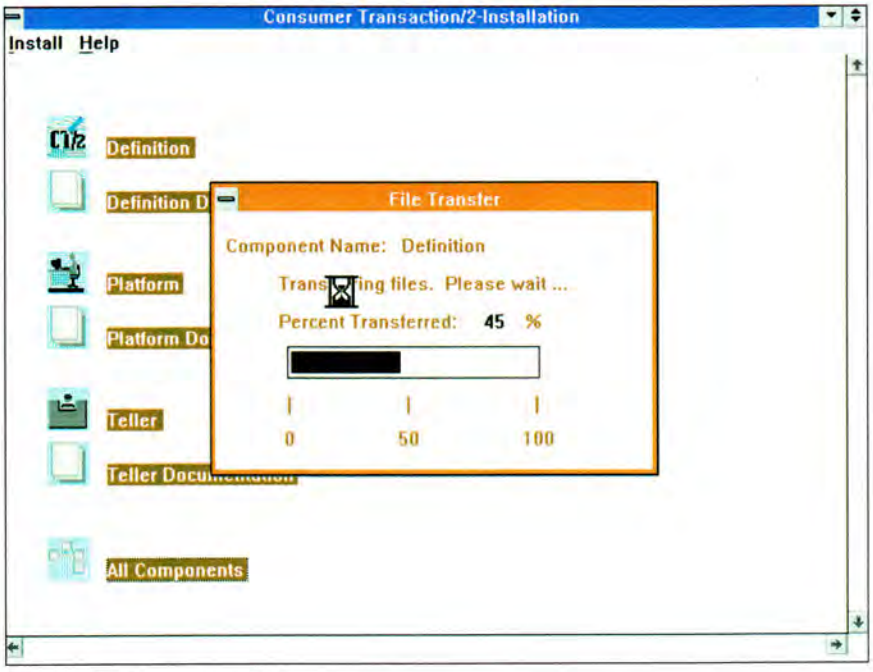


Figure 6. File Transfer Indicator

After the appropriate diskette has been inserted, the installation program creates the various subdirectories and begins its file copying procedure. Figure 6 depicts the "File Transfer" dialog box that is displayed to the user while the component files are being copied to their target location. This dialog box was developed in response to customer concern regarding the lack of appropriate feedback encountered in many installation programs, especially with respect to file transfer situations. Users resent, for instance, having to look at the disk drive light to discern whether an installation application is still in the process of copying files (one of the drawbacks of the previous installation program for OS/2). The "File Transfer" dialog box addresses this complaint by displaying the component currently being installed along with the percentage of files that have been

transferred so far. These percentages are presented in both a graphic and numeric format. Figure 7 represents a partial code listing for this "File Transfer" dialog box.

File Compression

One of the features of this installation program was its interface with PKWARE's file compression utility (PKZIP2). The advantages

```
MRESULT EXPENTRY DZZHFFTMMsgProc(hWndDlg, message, mp1, mp2)
HWND hWndDlg;
USHORT message;
MPARAM mp1;
MPARAM mp2;
{

***** Non-relevant code removed *****


/*********************************************************************/
/* Code for DosExePgm.  This API is used to issue the command to run */
/* the PKZIP ".exe" as a process in the background.                   */
/*********************************************************************/
  rc = DosExecPgm(achModuleName,
                  sizeof(achModuleName),
                  EXEC_ASYNCRESULT,
                  vszOptions,
                  NULL,
                  &resc,
                  vszExecPrgmParams);

***** Non-relevant code removed *****

  /*****************************************************************/
  /* While the PKZIP process is running, we need to detect how     */
  /* many files have been decompressed and copied (code not shown */
  /* here) and update the numeric and graphic indicators.  Keep    */
  /* updating until the PKZIP process has completed.               */
  /*****************************************************************/
  do
  {
    rc = DosCwait(DCWA_PROCESS, DCWW_NOWAIT, &resc,
                  &pidProcess, resc.codeTerminate);
    usSearchCount =1;

    /*****************************************************************/
    /* Update FilesTransferred to number of files unzipped.  Count- */
    /* NewFiles will read the filenames, with fullpath, from the     */
    /* ".lst" file and do a DosFindFirst for the file. This will     */
    /* be repeated until it fails to find a file or all the files    */
    /* in the ".lst" file are exhausted.  This function will return */
    /* the # of files successfully found.                            */
    /*****************************************************************/
    FilesTransferred = CountNewFiles( );
```

Figure 7. Code for File Transfer (Continued)

```
    for (i=0; i<30; i++)                                    (A)
    {
      while((WinPeekMsg(hAB, &qmsg, NULL, 0,0, PM_REMOVE)) == TRUE)
      {

       /*************************************************************/
       /* Look for messages and dispatch them.  This process will   */
       /* need to sleep so that other processes will get time to    */
       /* operate.                                                  */
       /*************************************************************/
       WinDispatchMsg(hAB, &qmsg);
       WinPostMsg(HWND_DESKTOP, WM_PAINT, 0L, 0L);
       DosSleep (7L);
      }
       /*******************************/
       /* Reset pointer to hour glass  */
       /*******************************/
      WinSetPointer(HWND_DESKTOP, WinQuerySysPointer(HWND_DESKTOP,
                    SPTR_WAIT, FALSE));

      DosSleep (1L);

    } /* end for loop */

***** Non-relevant code removed *****

    PctFilesTransferred = (FilesTransferred * 100) / TotalNumFiles;
    ltoa(PctFilesTransferred, PctFilesTransChar, 10);
    WinSetWindowText(WinWindowFromID(hWndDlg, 1005),PctFilesTransChar);

    /*************************************************/
    /* Create Bar Indicator                          */
    /*************************************************/
    strcpy (BarIndicator, "                    ");
    BarCounter =  PctFilesTransferred / 5;

    for (i = 20;  i > (SHORT)BarCounter; i --)
    {
      BarIndicator[i -1] = ' ';
    }

    if (i>0)
    {
      WinSetWindowText(WinWindowFromID(hWndDlg, 1007), BarIndicator);
    }

  }while (rc != ERROR_WAIT_NO_CHILDREN);

***** Non-relevant code removed *****
```

Figure 7. Code for File Transfer

of accessing an off-the-shelf, efficient file compression utility are two fold: First, the number of diskettes shipped for the product were reduced from 22 to 8. Second, since PKZIP2 is an already existing product, fewer resources were needed to write interface code and test the file compression routines. Because the installation program did not have access to the source code of PKZIP2, however, it needed to run the file compression routine as a separate process (i.e., as an ".exe" running in the background). This presented some problems in that PKZIP2 not only would control more processing time than necessary, but also blocked valid PM messages from reaching the message queue (e.g., Ctrl/Esc to multi-process, etc.). Section (A) in Figure 7 represents the code used to correct this situation. While the PKZIP2 process is active, this code checks the message queue to see if any messages are waiting. If so, it dispatches them and then "sleeps" for seven milliseconds to allow adequate time for other processes. If no messages are waiting, the "while" loop is ended and the program "sleeps" for only one millisecond. The graphic and numeric indicator windows are then updated accordingly. This program cycles through this section of code (counting the number of files copied, checking the message queue, and updating the indicator windows) until it receives a return code indicating that the PKZIP2 process has ended.

Updating the CONFIG.SYS

Once the installation program has completed installing all of the selected components, it then determines whether the CONFIG.SYS file needs to be updated. For this banking application, the CONFIG.SYS file is updated only if the Definition, Platform, or Teller components are installed. If one of these components is installed, the installation program checks to see if the CONFIG.SYS file existed, and if so, determines its file status (e.g., write-protected). If the file did not exist, the user is alerted and given the opportunity to save the changes to an empty file. This change file contains a complete listing of the paths to each of the selected components. If the CONFIG.SYS file did exist and was write-protected, the installation program asks for confirmation before it changes the CONFIG.SYS file status. The original CONFIG.SYS file is then opened and read intoa buffer. Each line in this buffer is then evaluated separately to determine if modification is necessary.

Figure 8 represents a partial code listing of the CONFIG.SYS updating procedure used to modify the SET PATH line. In addition to updating this line, the LIBPATH, SET HELP lines, and a series of environment variables

```
BOOL cwUpdateConfigSys(CallhWnd)

HWND CallhWnd;

{

***** Non-relevant code removed *****

/*********************************************************************/
/* The following segment of code tokenizes each line in the abBuffer */
/* (the original CONFIG.SYS) on the "\n" character and places it     */
/* into the "vszPathToken" variable.  It then executes the following */
/* WHILE condition as long as the vszPathToken variable is not equal */
/* to NULL or the end of file (EOF) character has not been detected. */
/*********************************************************************/

 vszPathToken = strtok(abBuffer, "\n");              (A)
```

Figure 8. Code for Updating the CONFIG. SYS File (Continued)

```
while ((vszPathToken != NULL) && (*vszPathToken != 0x1a))
{
 length = strlen(vszPathToken) - 1;
 /*****************************************************************/
 /* If the last character of the vszPathToken variable is the EOF   */
 /* character, then replace it with a carriage return.  This code   */
 /* is necessary to ensure that the installation program updates    */
 /* the CONFIG.SYS file properly.  That is, other applications      */
 /* often attach the EOF character to the end of the last line of   */
 /* this file.  The above WHILE condition would be unable to detect */
 /* this situation since it focuses on the first few characters to  */
 /* determine if its a valid line.                                  */
 /*****************************************************************/
 if(*(vszPathToken + length) == 0x1a)
 {
  *(vszPathToken + length) = 0x0d;
 }
 /*****************************************************************/
 /* Check to see if the vszPathToken variable contains the "SET     */
 /* Path" line of the CONFIG.SYS.  If so, set the PathUpdateFlag    */
 /* to TRUE and determine if this line needs to be modified.        */
 /*****************************************************************/
 if (strnicmp(vszPathToken, "set path=", 9) == 0)  (B)
 {
   PathUpdateFlag = TRUE;
   /***************************************************************/
   /* If the Definition flag is equal to TRUE, determine whether  */
   /* the contents of the vszSDOInfor[4] array element are already */
   /* contained in the vszPathToken variable (this is one of the  */
   /* system settings that was provided as a default or the user   */
   /* modified).  If it already exists, then copy contents of     */
   /* vszPathToken variable to a temporary buffer, change the file */
   /* pointer to the end of the file, then write the contents of  */
   /* this temp buffer to the file.  Otherwise, this line will need */
   /* to be modified.                                             */
   /***************************************************************/
   if (DefnFlag == TRUE)
   {
    strcpy(vszTemp1, vszPathToken);
    strcpy(vszTemp2, vszSDOInfo[4]);
    strupr(vszTemp1);
    strupr(vszTemp2);

    if (strstr(vszTemp1, vszTemp2) != NULL)
    {
     SetPathFlag = TRUE;
     strcpy(vsztmpBuffer,  vszPathToken);
     strcat(vsztmpBuffer, "\n");
     length = strlen(vsztmpBuffer);

     DosChgFilePtr(hfConfigMPH,
                   0L,
                   FILE_END,
```

Figure 8. Code for Updating the CONFIG. SYS File (Continued)

```
                &ulFilePointer);

    DosWrite(hfConfigMPH,
                vsztmpBuffer,
                length,
                &cbBytesWritten);

   }
  else
   {
    /*****************************************************************/
    /* The following lines of code attempt to insert the contents of */
    /* vszPathToken into the beginning section of the "SET PATH="    */
    /* statement.  This is done to optimize the performance of the   */
    /* component once the banking application is activated.  It does */
    /* this by copying the contents of the vszPathToken variable into*/
    /* a temporary variable, obtains its length and then reverses    */
    /* it. The "SET PATH=" text is copied into a second temporary    */
    /* variable and the components drive and directory (contained in */
    /* vszNewPathTmp) and a ";" is concatenated.  All but the "SET   */
    /* PATH=" text is copied from original "SET PATH" line, reversed */
    /* and concatenated to the vszTempToken variable.  This variable */
    /* is then copied to a buffer and the length of the line is      */
    /* assessed.  If it exceeds 256 characters (ASCII limit), the    */
    /* user is notified.  Otherwise, the file pointer is placed at   */
    /* the end of the file and the the contents of the buffer are    */
    /* copied to the file.                                           */
    /*****************************************************************/
    strcpy(vszTemp1, vszPathToken);                 (C)
    length = strlen(vszTemp1);
    strrev(vszTemp1);
    strcpy(vszTempToken, "SET PATH=");
    strcat(vszTempToken, vszNewPathTmp);
    strcat(vszTempToken, ";");
    strncpy(vszTemp2, vszTemp1, (length -9));
    vszTemp2[length-9] = '\0';
    strrev(vszTemp2);
    strcat(vszTempToken, vszTemp2);

    strcpy(vsztmpBuffer,  vszTempToken);
    strcat(vsztmpBuffer, "\n");
    length = strlen(vsztmpBuffer);

    if(length > MAXPATHLENGTH)
    {
      WinMessageBox(HWND_DESKTOP,
         CallhWnd,
         "Updates to the SET PATH line in the CONFIG.SYS file"\
         " will exceed 256 characters.  You will need to shorten"\
         " this line before the file can be modified.",
         "Consumer Transaction/2-Installation",
         0,
         MB_OK | MB_WARNING | MB_MOVEABLE);
```

Figure 8. Code for Updating the CONFIG. SYS File (Continued)

```
        return(FALSE);
      }

      DosChgFilePtr(hfConfigMPH,
                    0L,
                    FILE_END,
                    &ulFilePointer);

      DosWrite(hfConfigMPH,
                    vsztmpBuffer,
                    length,
                    &cbBytesWritten);

    } /* end if on strstr(vszTemp1, vszTemp2) != NULL */
  }
else
  {
    /*****************************************************************/
    /* If the Definition flag is not equal to TRUE, then the         */
    /* "SET PATH=" line should not be updated (requirement of        */
    /* application).  Therefore set the "SetPathFlag" to TRUE (used */
    /* later in the code to determine if the line was updated),      */
    /* copy the contents of the vszPathToken variable to a           */
    /* temporary buffer, change the file pointer to the end of the   */
    /* file, and copy the contents of the buffer to the file.        */
    /*****************************************************************/
    SetPathFlag = TRUE;
    strcpy(vsztmpBuffer,  vszPathToken);
    strcat(vsztmpBuffer, "\n");
    length = strlen(vsztmpBuffer);

    DosChgFilePtr(hfConfigMPH,
                  0L,
                  FILE_END,
                  &ulFilePointer);

    DosWrite(hfConfigMPH,
                  vsztmpBuffer,
                  length,
                  &cbBytesWritten);

   }/* end if on Defn Flag == TRUE */

  } /* end if on strnicmp(vszPathToken, "set path=", 9) == 0 */

***** Non-relevant code removed *****
```

Figure 8. Code for Updating the CONFIG. SYS File

(unique to the banking application) also are updated. As section (A) in this code listing points out, each line of the buffer is tokenized on the "\n" character and placed into a variable. As long as the first character of this variable is not the NULL or End of File (EOF)

character, the line is evaluated against a series of "if" statements to determine if it needs to be modified. Section (B) in this code listing highlights the "if" statement used to identify the "SET PATH" statement.

If one of the designated lines is detected, the installation program evaluates the line further to determine whether it contains a valid path to the selected component(s). Only those lines which do not contain a valid path are modified. Section (C) shows the optimization procedure used to modify the SET PATH line when a valid path is not present. This code is designed to place the path for the selected component(s) at the beginning, rather than the end of the path statement (a development specification to enhance the performance of the banking application). It then ensures that this line does not exceed 256 characters (ASCII limit) and writes it to a temporary buffer. Those lines which are not modified also are written to the same buffer. After updating the necessary lines, the original CONFIG.SYS file is renamed as a backup file, and the contents of the temporary buffer take its place.

Creating a Program Group and Title

The next task for the installation program is to create a program group and title within OS/2's Desktop Manager. Figure 9 represents a complete code listing for this procedure. When called, this procedure first creates a program group and obtains a handle. If a valid handle has been returned and the program title flag is true (e.g., user wants a program title to be created), the program title structure is initialized. Information for this structure is obtained from the "Change settings" dialog box. After initializing the structure, the program uses a WinQueryProgramTitle call to determine if the program title already exists. If so, it returns FALSE to the calling function which in turn alerts and enables the user to specify a new title. If the program title does not exist, and the CreateTitleFlag is TRUE, then the installation program will add the program title to the Desktop Manager.

The installation process is complete once it has successfully created a program group and title. A message box is displayed informing the user to select "Exit" under the "Install" pulldown in order to end the program. If the CONFIG.SYS file was modified, the user also is informed to reboot the machine prior to activating the banking application.

WHAT OUR CUSTOMERS ARE SAYING

Independent usability testing was conducted on the installation program once it entered the system test phase. Two separate studies were run to assess both usability and ease-of-use in

```
BOOL cwProgramGrpTitle(CallhWnd, CreateTitleFlag)

HWND CallhWnd;
BOOL CreateTitleFlag;

{
 static HPROGRAM hPrgGroup, hPrgTitle;
 static SHORT rc, i, pcTitles;
 static CHAR vszDefnExe[MAXPATHLENGTH];
 PPROGRAMENTRY prgtitles;

/*******************************************************/
/* Send PM call to create Program Group.               */
/*******************************************************/
  hPrgGroup = PrfCreateGroup(HINI_USERPROFILE, vszSDOInfo[1],
                             SHE_VISIBLE);
```

Figure 9. Code for Creating a Program Group and Title (Continued)

It was the most professional, easiest, and most clear of the three programs I installed

```
/*************************************************************/
/* Create structure for Program Title only if you get a handle  */
/* to the Program Group and the Program Title flag == TRUE      */
/*************************************************************/
  if  ((hPrgGroup) && (ProgramTitleFlag == TRUE))
  {
   PROGDETAILS progde;  // Initialize Prg Title structure

   memset(&progde, '\0', sizeof(PROGDETAILS));
   progde.pszTitle = malloc(MAXNAMEL);
   progde.pszExecutable = malloc(MAXPATHLENGTH);
   progde.pszStartupDir = malloc(MAXPATHLENGTH);
   progde.pszIcon = malloc(MAXPATHLENGTH);
   strcpy(vszDefnExe, CheckSlash(vszSDOInfo[4]));
   strcat(vszDefnExe, "dzzdo.exe" );  // name of Definition EXE file
   progde.Length = sizeof(PROGDETAILS);
   progde.progt.fbVisible = SHE_VISIBLE;
   progde.progt.progc = PROG_DEFAULT;
   strcpy(progde.pszTitle, vszSDOInfo[5]);
   strcpy(progde.pszExecutable, vszDefnExe);
   progde.pszParameters = NULL;
   strcpy(progde.pszStartupDir, vszSDOInfo[4]);
   strcpy(progde.pszIcon, vszSDOInfo[1]);
   progde.pszEnvironment = NULL;
   progde.swpInitial.x = 0;
   progde.swpInitial.y = 0;
   progde.swpInitial.cx = 0;
   progde.swpInitial.cy = 0;
   progde.swpInitial.fs = SWP_SHOW;
   progde.swpInitial.hwndInsertBehind = HWND_TOP;
   progde.swpInitial.hwnd = HWND_DESKTOP;

/************************************************/
/* Query whether the program title already exists. */
/* If so, return FALSE to the calling function.    */
/************************************************/
   prgtitles = malloc(20*sizeof(PROGRAMENTRY));
   rc = WinQueryProgramTitles(hAB, hPrgGroup, prgtitles, 512,
&pcTitles);

   if (pcTitles > 0)  // Determine if title exists
   {
    for(i = 0; i < pcTitles; i++)
    {
     if(stricmp(prgtitles->szTitle, vszSDOInfo[5]) == 0)
     {
      return(FALSE);
     }
     prgtitles ++;
    }
   }
```

Figure 9. Code for Creating a Program Group and Title (Continued)

```
/*****************************************/
/* Add new program title.                */
/*****************************************/
   if (CreateTitleFlag == TRUE)
   {
    hPrgTitle = PrfAddProgram(HINI_USERPROFILE, &progde, hPrgGroup);
   }

  } /* end if on hProg handle &Title Flag == TRUE */

  return (TRUE);

} /* end Program group/title procedure */
```

Figure 9. Code for Creating a Program Group and Title

relation to installation programs for two main banking competitors. Both bankers and their IBM system engineers took part in the testing. In all, 27 people from 6 banks participated in the studies.

A number of different installation scenarios were tested. Even with the most complicated of installation procedures, where only a subset of components were installed, the default disk drives and paths were changed, and modifications were made to the Program Group, each one of the test participants was able to install the programs on the first trial. This compares with only a 23.1% first installation success rate for the predecessor DOS version.

A second study was conducted to determine how this installation program compared with that of the installation programs of two major competitors. The results show that 78% of test participants preferred this installation program over one of the competitors and 100% preferred it over a second competitor. The first competitor installation program did not allow the user to select the number of components to be installed. With that installation program the bank was forced to install components that may not be needed.

Written comments indicated that test participants evaluated the installation program quite favorably:

- "Simple and user friendly"
- "Basically you can just click on things and let it run itself"
- "The manual is not even needed"
- "There were a lot fewer diskettes than I expected!"
- "It was the most professional, easiest, and most clear of the three programs I installed."
- "Compared to the other competitor products I installed, this one was by far the best. I hope all IBM products conform to these well-designed processes."

It is sometimes said that first impressions are everything. That may be an overstatement, but in fact, first impressions play an important role in helping to form early judgments about both people and things. The first impression a user forms about a software product typically occurs in using the product's installation program. An easy-to-use, consistent, and intuitive installation program is a big first step toward making a good impression with your software customer. Hopefully, this article provides some helpful hints for making that first big step.

An easy-to-use, consistent, and intuitive installation program is a big first step toward making a good impression with your software customer

REFERENCES

IBM CUA '91 Advanced Interface Design Reference (SC34-4290-00).

Michael Heck, *IBM Corporation, 1001 W.T. Harris Boulevard, Charlotte, NC 28257. Mr. Heck is a research engineer/scientist with IBM specializing in Human Factors. He holds an MS degree in Personality/Social Psychology from Kansas State University and is currently pursuing his Ph.D. in Applied-Experimental Psychology from the same university. Mr. Heck has spent much of the past year programming various applications within the PM environment.*

James Rudd, *IBM Corporation, 1001 W.T. Harris Boulevard, Charlotte, NC 28257. Dr. Rudd joined IBM as a human factors scientist in 1988. He earned a Ph.D. in applied psychology from Virginia Polytechnic Institute and State University in 1986 and is currently pursuing an MS in computer science at the University of North Carolina — Charlotte. Prior to joining IBM, Dr. Rudd worked for The Aerospace Corporation as a member of the technical staff. Currently he is prototyping object-oriented user interface concepts for finance industry applications.*

International Business Machines Corporation
P.O. Box 1328
Boca Raton, FL 33429-1328

G362-0001-12